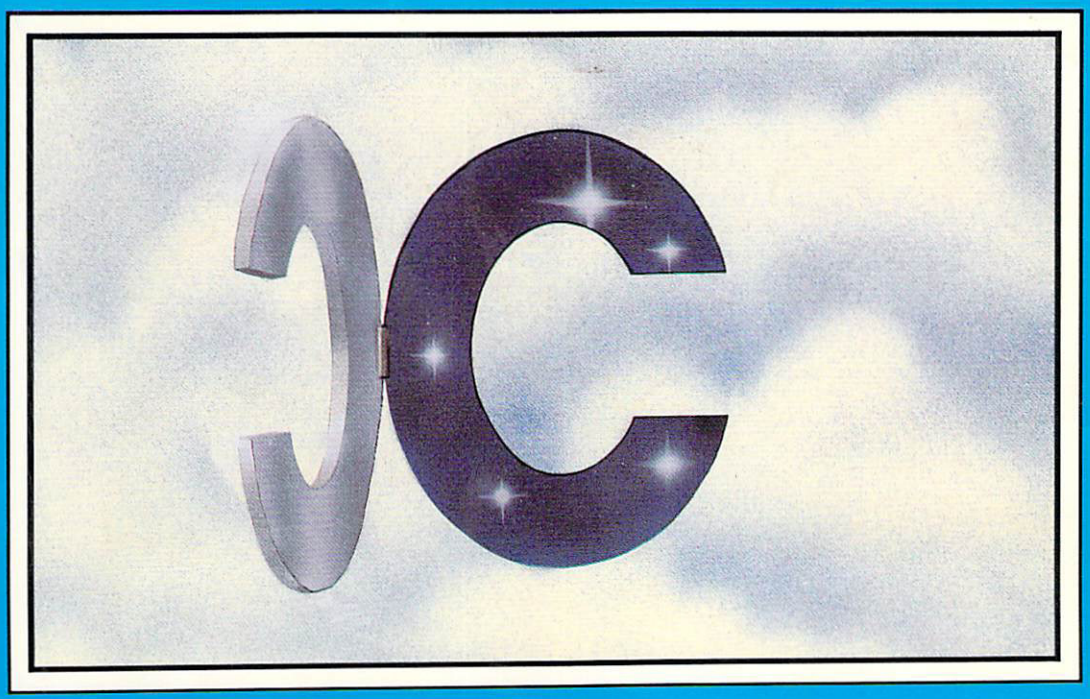
Amiga® C for Beginners

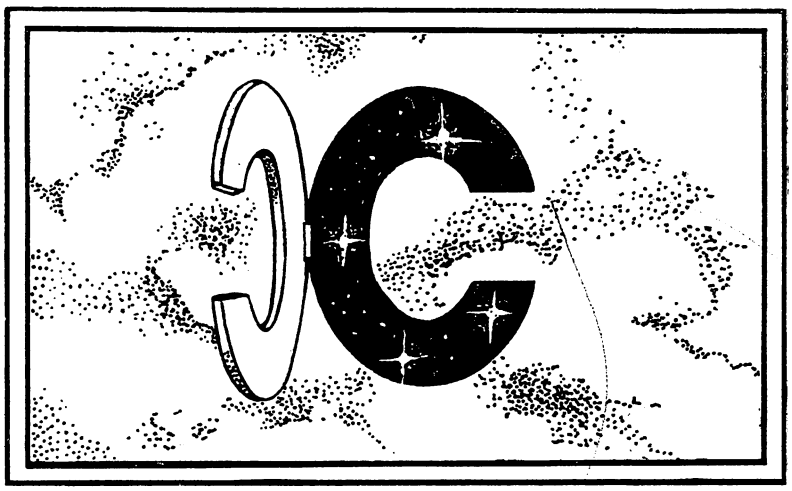
A Practical guide to learning and using C Language on your Amiga

by Dirk Schaun



Abacus
A Data Becker Book

Amiga
C for Beginners



Dirk Schaun

Abacus
A Data Becker Book

Fourth Printing, 1990
Printed in U.S.A.
Copyright © 1989, 1990
Abacus
5370 52nd Street, SE
Grand Rapids, MI 49512

Copyright © 1987, 1988
Data Becker GmbH
Merowingerstrasse 30
4000 Duesseldorf, West Germany

This book is copyrighted. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of Abacus or Data Becker GmbH.

Every effort has been made to ensure complete and accurate information concerning the material presented in this book. However, Abacus can neither guarantee nor be held legally responsible for any mistakes in printing or faulty instructions contained in this book. The authors always appreciate receiving notice of any errors or misprints.

AmigaBASIC is a trademark or registered trademark of Microsoft Corporation. Amiga 500, Amiga 1000, Amiga 2000, Amiga and C64 are trademarks or registered trademarks of Commodore-Amiga, Inc. Lattice C and Lattice are trademarks or registered trademarks of Lattice Corporation. Aztec C and Aztec are trademarks or registered trademarks of Manx Software Systems. IBM is a trademark or registered trademark of International Business Machines, Inc. Atari ST is a trademark or registered trademark of Atari Corporation.

ISBN 1-55755-045-X

Table of Contents

1. Introduction to C 1
1.1 Program Execution 4
1.2 Compiler vs. Interpreter 5

2. Beginning C 7
2.1 The Editor 10
2.2 The Compiler 11
2.3 The Linker 12
2.4 Putting It All Together 13

3. The First Program 15
3.1 Using ED 18
3.2 Compiling 19
3.3 Error Messages 21

4. Theory and Practice 23
4.1 Program Format 27
4.2 Defining a Function 28
4.3 printf and Escape Sequences 29
4.4 Comments 30
4.5 Variables and Arithmetic 31
4.5.1 Integers 31
4.5.2 The if statement 32
4.5.3 Calculating with C 35
4.5.4 Floating Point Numbers 37
4.5.5 Characters and Character Strings 39

5. Loops 43
5.1 while Loops 45
5.2 for Loops 48
5.3 do while Loops 49
5.3.1 More Error Checking 49
5.4 AND and OR 52

6. Strings 55
6.1 Backtracking 57

7. Calculating in C 59

8. Variables 63
8.1 Variable Names 65
8.2 Data Types 67
8.3 Type Conversion 70
8.4 The cast Operator 71

9. printf and scanf 73
9.1 More Escape Sequences 75
9.2 Format Specification 77
9.3 Octal and Hexadecimal 80
9.3.1 Conversion Program 82
9.4 Character Codes 84
9.4.1 About the Backslash 85
9.4.2 Going the Other Direction 86

10. The Preprocessor 87
10.1 #define 89
10.2 #include 91

11. Abbreviations 93
11.1 Increment and Decrement 97
11.2 Initialization, Definition, Declaration 99
11.3 Multiple Assignments and Directive Value in C 101

12. Functions 103
12.1 Functions with Arguments 106
12.2 Functions without Return Values 108
12.3 Other Functions 109
12.3.1 strcpy-Version 1 109
12.3.2 strlen 111

13. Arrays 113
13.1 Multi-dimensional Arrays 116

14. More about Loops 121
14.1 More about the for Loop 123
14.2 break 124
14.3 continue 125
14.4 The switch Directive 126

15. Pointers and Addresses 129
15.1 Addresses 131
15.2 Pointers 133
15.2.1 The Exchange Function with Pointer 134
15.2.2 strcopy-Version 2 135
15.2.3 strcopy-Version 3 135
15.3 Pointer without Storage 138

16. Storage Classes 141
16.1 Auto 143
16.2 Static 144
16.3 External 145
16.4 Register 146
16.4.1 Fast strcpy Routine 147
16.5 Local 149

17. User-defined Libraries 151
17.1 The strcmp Function 154
17.2 Itoa 157
17.3 Reverse 159

18. C Features 161
18.1 The ?: Operator 163
18.2 The sizeof Function 164
18.3 Bit Manipulation 165
18.3.1 AND 165
18.3.2 OR 166
18.3.3 Bitwise Shift Operators 167
18.3.4 EXCLUSIVE OR 168
18.3.5 One's Complement 168
18.4 Goto 169

19. Complex Data Types 171
19.1 Struct 173
19.2 Bit Fields 175
19.3 Unions 176
19.4 Enum 177
19.5 Typedef 178

20. Important Concepts 179
20.1 Declarations 181
20.2 Initialization 184

21. Pointer Arrays 187

22. Useful Macros 193
22.1 Macro Error Sources 196
22.2 Library Macros 198

23. Communication 201
23.1 Passing Data with the CLI 203
23.2 Buffered Input/Output 206
23.3 More Buffered Input/Output 211
23.4 Unbuffered Input/Output 213
23.5 Direct Access 216
23.6 Reading a Character 218
23.6.1 Standard Input/Output 218
23.7 A User Window 219
23.7.1 The Three Windows 219
23.8 Redirection 222

24. Tricks and Tips 225
24.1 Starting from the Workbench 227
24.2 Other Preprocessor Directives 229
24.3 Finding and Removing Errors 231

25. System Programming 233
25.1 The Intuition Principle 235
25.2 A Window under Intuition 236
25.2.1 The Window Flags 237
25.2.2 Opening a Window 238
25.2.3 A Window Program 239
25.3 Screens 242
25.3.1 A Screen Program 243
25.4 Text/Graphic Window Display 246
25.4.1 `Text` 246
25.4.2 `Move` 246
25.4.3 `Draw` 247
25.4.4 Small Drawing Program 248
25.4.5 Low Resolution and Interlace Modes 252
25.4.6 Pixel Processing 253
25.5 DOS 257
25.6 `SetComment` 258
25.7 Read Directory 259
25.8 Conclusion 262

Appendices 263
A. Functions 265
B. The History of C 270
C. The Lattice C Compiler 272
D. The Aztec C Compiler 273
E. Reserved C Words 276
F. Operator Precedence 277
G. Storage Classes 278
H. Type Conversions 278
I. Modes for `fopen` 279

Index 281

1
Introduction to C

1. Introduction to C

So you want to learn C. That's obvious, or you wouldn't be reading this book. The goal of this book is to help you learn to program in C on the Amiga in as little time as possible. *Amiga C for Beginners* is written as a short course in the C language for anyone who wants to learn about the C language from scratch, without a lot of technical jargon.

This book is divided roughly into two parts. The first part introduces the reader to the basic structures of C programming through operating the editor, compiler and linker. It also contains many sample programs. You'll be able to write your own short programs in C in only a few hours.

This first part also takes the beginning C programmer through the essentials of C—calculations, string handling, loops and more. It even helps you find the errors made most often by new C programmers.

The second part, which begins at Chapter 10, explains the background and peculiarities of each C statement and function. These include preprocessor commands, arrays, loops, pointers, addresses and memory classes. If these words don't mean anything to you now, they will when you start reading the second section.

The second part also discusses macros, interfacing your Amiga to the outside world, tricks and tips for the C language and graphic programming.

The appendices describe the history and development of the C language, as well as operating instructions for a number of popular C compilers (see your compiler's instruction manual, since many compilers are constantly being upgraded). In addition, you'll find a number of practical C functions, mathematical precedence, reserved C keywords, memory classes and type conversions.

This chapter describes the general nature of C. You'll see its advantages and disadvantages compared to interpreted languages, as well as advantages and disadvantages compared to other compiled languages. You'll also see why C stands out above so many other computer languages.

1.1 Program Execution

Before starting we need to ask the big question: "What is C?"

There are two kinds of computer languages. First, there are interpreted languages such as BASIC and LOGO. Second, there are compiled languages such as C, Pascal or Modula2. Section 1.2 contains detailed information about the advantages and disadvantages of interpreters and compilers.

Compilers

Compilers are programs which translate the language statements into a form understandable to the computer. This form consists of the numbers 0 and 1 (the numbers used in the binary system). Since people can't remember long strings of zeros and ones, the computer can be told to interpret words and other number systems as binary numbers. For example, a typical machine language instruction LDA means "Load the Accumulator", which is easier to remember than the binary number 10011101.

The computer must contain a central processing unit (CPU) to respond to the machine language instruction available in response to every keyword. Programming with the binary numbers is called machine language programming. Using machine language mnemonics or instructions is called assembly language. The instruction which executes on one computer may not even exist on another computer.

It would be easier for the programmer if the computer could be told in plain English what to do. Needless to say, computer languages haven't reached the level of a DWIM (Do What I Mean) interface. This type of language may exist a few years from now, when technology produces more helpful developments using artificial intelligence.

High level languages act as a compromise between machine language and human language. These languages contain a limited number of statements or keywords which in turn execute specific tasks. Unlike assembly language, many of these languages aren't tailored to a specific computer. The computer must execute several hundred machine instructions for every single statement in the high level language. An example would be the BASIC command "`LOAD Filename`", which tells the computer to load a file.

1.2 Compiler vs. Interpreter

An interpreter

As previously mentioned, two types of higher level languages exist: Interpreted and compiled. An interpreter searches for keywords in the program text (or source), verifies that the command is a legal one and executes the equivalent machine language instructions. Then the interpreter searches for the next command, tests it and executes the machine instruction, etc.

This translation process performed by the interpreter can be compared to the work of a human foreign language interpreter. An interpreter is a translator which translates the words of the higher level language into machine level instructions as needed.

A compiler

A compiler translates the source program once into executable form. This is similar to someone who translates foreign language literature into his/her own language. This translator takes the time to select the proper choice of words for the text, unlike the interpreter who mediates a conversation between two people from different language backgrounds.

The interpreter executes programs immediately. The interpreter can also be stopped by the user to check on certain values stored in certain variables. Interpreted program execution can usually be continued without causing problems. The main advantages of interpreted languages are flexibility and spontaneity.

The compiler translates the program source only once. The compiler may spend a few minutes compiling the program before it can be executed. Afterwards, the compiled program executes much faster than an interpreted program, since the compiler doesn't have to re-translate the source program. This advantage becomes most evident in program loops in which a command can be executed several thousand times. The interpreter translates the command into machine language, repeating this process a few thousand times. The compiled program already knows what to do without repeating the translation. The advantages of compiled languages lie in shorter execution times for compiled programs.

Interpreters, because of easy access to their programs, let the programmer enter corrections and modifications on the fly. They also let the programmer enter and execute program code until an error occurs. This "run-until-it-breaks-then-fix-it" attitude causes lazy, unstructured programming style.

Compilers don't usually allow easy error correction. When an error occurs, you must reload the program source editor, fix the errors in the source code and recompile the program. In addition, compiled languages require a specific language structure, or the program code won't compile. The source code must be correct from beginning to end or the compiler will not compile it.

The biggest advantage that C has over other languages is its transportability. This means that you should be able to take a C source code from an Amiga, transfer it to a PC, make changes to fit the PC's file handling and other machine-specific tasks, and compile the C source code on the PC with no problem.

There you have a general overview of the advantages and disadvantages of compiled languages and interpreted languages. The next chapter spends some time talking about these languages as well, while adding specifics about the subject of this book—the C language.

2.
Beginning C

2. Beginning C

Let's look at the process involved in developing a C program. Although it may seem long and involved, don't panic. Once you learn the basics of developing a high level language program, it gets easier.

One word of warning: Don't skip this chapter, even if you have previous experience with C. You might learn something you didn't know before about the language.

Programs begin with an idea. The user has a task that he wants to accomplish using a program—and the program should perform the task easier, faster or more accurately than a human could do it. This idea could be for a drawing program; a spreadsheet for calculating payroll and figures; a fast disk copier; or just a simple text display on the screen.

Once the general idea is developed, it helps if the programmer sits down and writes out the goals of the idea, and how the program can do this. This writing stage can be in plain English, since it should be as readable for you as possible. This written documentation of the program execution is sometimes called the pseudo code, since it tells what the program should do without actually writing which statements the program needs to perform the task. When writing the pseudo code of this program, keep it broken down into smaller modules whenever possible.

A data flowchart and program flowchart should be developed from the written documentation of the idea, just as in any other computer language. After it has been determined how the program flow should appear, the user can proceed to the computer to program in the idea.

The C language uses three different programs which work together in generating programs:

- The editor, in which the user enters and corrects the source program;
- The compiler, which compiles the program;
- The linker, which joins the main program with other compiled programs and functions to make a fully executable program.

The rest of this chapter discusses the use of each program in the development of C programs.

2.1 The Editor

You need some sort of text editor to enter a C program from the keyboard. An editor is nothing more than a simple word processor. It usually contains only minimal text processing capabilities. The program usually only allows you to type in, load, save and edit the text of the source code: Nothing fancy like block functions or save and replace capabilities.

Most word processor programs can be used for typing in the text of a C program. If you use a word processor, you may not enter any special control characters (e.g., bold fonts and text formatting) because the C compiler would not recognize them. Many word processors provide the option of letting you save a file as an ASCII file.

If you use a word processor as an editor, it must allow you to enter the special control characters needed by C source codes. C programs use braces ({}), brackets ([]), the backslash (\), the number sign (#), the pipe character (|) and the tilde character (~).

Maybe you can't afford a word processor, or maybe your word processor doesn't have the necessary characters. The Workbench disk which comes with your Amiga contains an editor named `ED`. `ED` is a basic text editor, which you can find on the Workbench disk of the Amiga from the `CLI`. Invoke the editor by entering the `CLI`, typing `ed` and the name of the file you want to load/edit, and press the <Return> key.

The editor is loaded and then the C program is typed in. This text, called C source code, is stored on the disk under a filename. The characteristic that sets this apart from normal word processing files appears in the file extension: C source codes must contain a file extension of .C. Examples of names can be `sort.c` or `archive.c`; note that these names indicate the contents of each file. Try to use meaningful names like this instead of cryptic names like `a.c` or `this.c`. The extension of .C is especially important, since several files with the same name but different extensions are generated by a C compiler. After a file is created and saved, you can then call the C compiler to compile the C source code.

2.2 The Compiler

Calling the compiler loads the C compiler into memory. The compiler reads the source code and begins to convert the source code into compiled machine language. Most compilers read the source code twice (two pass compiler). During the first pass, if it encounters an invalid expression or keyword, it stops compiling and displays an error message on the screen or writes the error to a disk message file.

Whenever an error is found, you must reload the editor, and correct the source code. Once you've fixed the errors, you save the source code file, exit the editor and restart the compiler. If the compiler finds errors again, you must repeat the above procedures.

Before going any farther, you should realize one thing. Writing the C source code requires proper preparation. If you just enter source code without giving any thought to what you're typing in, you'll spend more time learning about error messages and the editor than you will spend learning about the C language. In the beginning, prepare to see plenty of errors, and be patient with yourself as you go from compiler to editor to compiler.

When you've corrected all of the errors in the source code, the compiler can finish the first pass without stopping and performs the second pass. This second pass does the final transformation into *object* (compiled) code. The compiler saves the object code to disk under the same name as the source code, but with an extension of .O. For example, a source code named `source.c` generates an object code file named `source.o`. The object file needs one more step before it can become executable program code—this is the linker.

2.3 The Linker

We're not done quite yet. The object file must still be run through the linker. The linker searches for all functions used by the program from the C libraries, and links the necessary functions into one program.

Functions

A function is a subroutine similar to a procedure in Pascal. Functions are capable of solving small tasks such as drawing a line or displaying a character on the screen. The libraries contained in most C compiler packages include frequently used functions stored in object (compiled) form. These functions could be input/output functions, graphic routines, sound routines or even trigonometric functions.

The linker identifies the functions required for a complete program and adds them to the main program. This saves a programmer a lot of work. The functions simply require the passing of values instead of retyping the source code for each function from scratch.

The C linker permits the development of large programs in modular form. This means that several parts can be (and often are) developed separately. The user can compile and test every module separately. This has the advantage that the complete C program doesn't have to be loaded and recompiled every time the compiler detects an error. The linker eventually links all the compiled modules into one complete, executable C program.

The linker cannot handle non-compiled source code. It can only link compiled functions together. Source codes can load other source codes during compilation, but we'll see more on this later.

2.4 Putting It All Together

The CLI

The Amiga's CLI (Command Line Interface) is used to specify the parameters for the editor, the C compiler and the C linker. For example, entering the following line in the CLI calls the Lattice C linker:

```
ALINK file1.o file2.o TO complete
```

You already know that a program may have to be compiled several times before it is free of errors (syntax errors, not logical errors). You must enter the above line exactly as written. Entering the line incorrectly can cause errors in itself. The developers of most C compilers took this into consideration, and added a special feature to allow easy linker access from text files.

The MAKE file

The inputs required to compile and link a C source code can be written to a file called a MAKE file. This MAKE file calls all the necessary programs such as the compiler or linker. The C system reads the file just as if the user had input the text direct from the keyboard.

It's easy to create a MAKE file. Instead of executing, the C compiler calls directly. You invoke the editor and write the calls to a script file. Once you save this file to disk, you now have a MAKE file.

AmigaDOS' Execute command reads this file and passes the information to the C compiler sections needed to make the final executable program. See Appendix C for one example of a MAKE file and its contents.

3.
The First Program

3. The First Program

The following code is our first C language program. Don't enter it yet—you'll type it in a few minutes starting at Section 3.1. Here's the source code so you can see what it looks like:

```
#include <stdio.h>
void main()
{
  printf("Hello, I am here!");
}
```

To see what the program produces, the text must be entered using an editor. When you use the editor, please enter the text exactly as it is printed here and in Section 3.1. This avoids error messages which will cause problems. Once you've become more comfortable working in C, you can change programs around to suit your own needs. But don't change anything until after the program compiles and links exactly as you see here.

A step by step procedure follows. If you're on the Workbench, you must start the CLI before anything else. The CLI can be found in the System drawer located on the Workbench disk. Workbench 1.3 users can use the shell program, which is an enhanced version of the CLI.

Workbench 1.2 users who can't find the CLI may have it switched off with the settings in Preferences 1.2. Using Preferences 1.2, you must click on the ON gadget next to the word CLI. Once you close and reopen the Workbench disk icon, the CLI icon should appear in the window.

After you invoke the CLI or shell, a new window appears. This window prompts for an input with the message:

```
1>
```

For a super computer such as the Amiga, this program is something unusual. No icons appear, the mouse can only be used to move and size the window, and the CLI only accepts input from the keyboard. Use the mouse to enlarge the CLI window to its maximum size. Everything that occurs from now on will be displayed in the CLI window. Now put the mouse aside—you won't be needing it for a while.

3.1 Using ED

First you'll need an editor to enter the program. The ED editor can be called from the C: directory on your Workbench disk. The following executes ED and creates a new file named hello.c:

```
1> ED HELLO.C
```

The computer places the 1> prompt at the beginning of the line. Some versions of the CLI may also display the current directory (e.g., the shell from Workbench 1.3).

The name hello.c will be the name of our first C language program. It doesn't matter whether you type the name in uppercase or lowercase letters.

If you made a typing error, press the <Backspace> key to delete the last character typed. The <Backspace> key has an arrow pointing to the left on some versions of the Amiga. If you prefer, you can press the <Ctrl><X> key combination to delete the entire line of text.

Press the <Return> key to execute the command and invoke the editor. A window appears and displays the text "Creating new file." The user now enters the program. The editor allows you to move the cursor around the file using the cursor keys to make corrections and changes. Type the following text:

```
#include <stdio.h>
void main()
{
 printf ("Hello, I am here!");
}
```

Once you've finished typing the text, press the <Esc> key, then press the <S> and <A> keys. Pressing the <Return> key saves the text (this combination will be called <Esc><SA> from here on). Press <Esc><X><Return> to save the text and quit the editor. The system returns you to the CLI. Pressing <Esc><Q> returns you to the CLI without saving the file. See the book *AmigaDOS Inside and Out* from Abacus for more information about ED.

Pressing <Esc><X><Return> returns the user to the CLI window immediately. You now have your first C source code ready to compile.

3.2 Compiling

Lattice 4.0

Start the Lattice 4.0 compiler with the following:

```
1> lc -L hello
```

For Lattice C 4.0, the following should now appear on the screen. Your screen may differ slightly. If you get error messages, see Section 3.3.

```
1> lc -L hello
Lattice Amiga DOS C Compiler Version 4.0
Copyright (C) 1987 SAS Institute Inc. All rights reserved

Compiling hello.c
Module size P=00000014 D=00000012 U=00000000
Total files: 1, Compiled OK: 1

Linking hello
BLink - Version 7.2
Copyright (C) 1986 The Software Distillery.
Copyright (C) 1987 SAS Institute Inc. All rights reserved
Box 8000 SAS Circle, Cary NC 27511-8000 - Telex 802505 (919) 467-8000


Blink complete - Maximum code size = 5488 ($00001570) bytes

Final output file size = 5312 (000014c0) bytes
1>
```

Aztec C

To compile the program using the Aztec C compiler requires two steps. The Aztec system first compiles the source code then assembles and links it. A MAKE file can be quite useful. Enter the following for the Aztec C compiler:

```
1>cc +L hello
```

The above sequence may not function in some cases. If not, enter the same line but omit the +L. Aztec C should display the following on the screen. Your screen may differ slightly. If you get error messages, see Section 3.3.

```
1> cc +L hello
Aztec C68K 3.6a 12-18-87 (C) 1982-1987 by Manx Software Systems, Inc.
Aztec 68000 Assembler 3.6a 12-18-87
1>
```

Now enter:

```
1>ln hello.o -lm -lc
```

Aztec C should display the following on the screen. Your screen may differ slightly.

```
1>ln hello.o -lm -lc
Aztec C68K Linker 3.6a 12-18-87
Base: 000000 Code: 001444 Data: 0002a0 Udata: 000050 Total: 001734
1>
```

Success

Did everything work as expected? If you didn't get an error, type the `dir` command in the `CLI` to see the current disk's directory. The executable program is stored there under the name `hello`. Notice that this file has no extension. There may also be other files with extensions of .MAP, .O, .LNK and of course .C. This shows that the extension helps identify the file. Call the executable program by entering the following line:

```
hello
```

On the screen appears the text:

```
Hello, I am here!
```

The program isn't earth-shattering, but this is only the beginning.

3.3 Error Messages

The most common mistake a new C programmer makes is omitting the semicolon following the closing parenthesis of the `printf` function. This semicolon is one of the most widely used characters in C programs since it indicates the end of a statement. For this reason almost every C statement or function ends with a semicolon. If the user omits it, the compiler reports many error messages. You might like to edit the `hello.c` file again and remove the semicolon. Save the file and try compiling the file again.

Lattice

If you forgot to type in the semicolon following the `printf` function, the Lattice C compiler displays the following message:

```
hello.c 5 Error 57: semi-colon expected
Compiler returncode 1
```

Aztec

The Aztec C compiler displays the following message if the semicolon is missing:

```
hello.c:5 ERROR 69: missing semi-colon
1 errors
```

Let's try to determine from this compiler message what is wrong with the file. The first line states the filename in which the error appeared: `hello.c`. That filename specification is useful later on (more on this later). Then follows the line number (5), error number (57) and the error description in English. This indicates that a semicolon was expected in line 5.

The user must now reload the editor to correct the error:

```
ED HELLO.C
```

You must include the extension of .C to edit the C source code.

To reach line 5, you can count down the lines (the fastest method for a program this short). You can also press <Esc><M><5><Return> to get to line 5. The editor then moves the cursor to the line indicated.

Line 5 consists only of the closing brace (}), but this is expected since the error actually occurred in the previous line. The semicolon in the previous line is missing. The error messages of the C compiler should never be taken too literally, since the search for the error may have to take the surrounding code lines into consideration.

After placing the semicolon where it belongs (following the `printf` function), save the file again and try to compile the source code again. It should work.

4.
Theory and Practice

4. Theory and Practice

Now that you've had some practical experience entering and compiling a program, let's look at the theory of how the program in Chapter 3 works.

You have two different types of keywords involved in C programming: functions and statements. The first line contains the function name `void main`. The `main` function is the most important element of a C program. The `void` means that this function will not return a value and is inserted so that the compiler does not display a warning message. Without this function, literally nothing runs.

Functions

A C program usually consists of up to a hundred functions. A function handles a part of the complete program, and is marked by parentheses (). Braces ({}) mark the beginning and end of the function. These braces surround the statements and functions which the computer should execute. The following lines call functions:

```
printf("Hello");
Value(10);
music();
end();
```

and the following lines don't:

```
value = old;
music;
end;
```

If you haven't compiled and linked the program in Chapter 3, do so now. When you execute the program from the `CLI`, a jump occurs first to the `main()` function. This always happens, regardless of where the function appears in the listing. It can be at the beginning, middle or end of the program, but the `main()` function always executes first. The `main` function calls the `printf` function. The `printf` function is stored in a library. All the user needs to know is the function name (`printf`), what it does (displays text on the screen) and what information it requires (text).

Arguments

The information passed to a function during the call are arguments. The arguments to be passed are placed within the calling function's parentheses to ensure proper delivery. It is important to enclose any character strings within quotation marks (e.g., `printf("Here I am!");`).

After calling the `printf` function the text appears on the screen. The `printf` function ends and the program continues at the point where it was interrupted by the function call.

No other statements follow the line after `printf`. This means that the `main` function has also reached its end, and the program ends. The computer returns to the `CLI`, and additional commands can be entered.

The end of the program and the return to the `CLI` represent the termination of the `main` function. The user should note how important this function is. It represents the C program itself; program execution starts and ends with the `main` function.

4.1 Program Format

Let's discuss the format of the program. The C compiler ignores any spaces, linefeeds and end of paragraph marks added to the listing by the user. Indenting lines or adding blank lines helps make source codes more readable to the user. Formatting has no effect on the execution speed or the length of the final program. The program you entered in Chapter 3 could have been in one of the following formats:

```
void main (){
     printf("Hello, I am here!");
     }

void main ()
{
     printf("Hello, I am here!");
     }

void main (){printf("Hello, I am here!");}
```

It's up to the user to select the version which appears to be most readable. Once you've selected a style, stick with it. However, the last version above illustrates how even a small program can be made unreadable by "formatting" it.

Note: All statements and functions must be entered in lowercase since C is case-sensitive (it differentiates between upper and lowercase letters). If you entered `Printf` or `PRINTF` instead of `printf`, the linker reports an error since it cannot find this function anywhere.

4.2 Defining a Function

A function only executes the statements contained within the braces. If nothing is written there, the computer does nothing. A program that does nothing is not very exciting, but it's a good example. The following C source code compiles without problems (and does nothing):

```
void main()
{}
```

Don't expect miracles from the above program. When you compile, link and start it, it loads, runs and does nothing. The computer returns to the CLI.

Definition

The function arguments enclosed in the braces are its definition. It defines what the computer should do when it executes a certain function.

A function can contain several statements or other function calls. Longer lines of text can be displayed on the screen with this program:

```
void main()
{
 printf("Hello, I have a question!\n");
 printf("Do you believe in life without electricity?\n");
 printf("Not me!\n");
}
```

The screen output will appear as follows:

```
Hello, I have a question!
Do you believe in life without electricity?
Not me!
```

This program has two differences from the program in Chapter 3. First, there are three `printf` functions instead of one; second, the `\n` character appears at the end of each string within quotation marks. See the next section for details on `\n` and other escape sequences.

4.3 printf and Escape Sequences

The text at the end of Section 4.2 appears on the screen as it appears in the source code, with three exceptions. The three `\n` characters do not appear. The `\n` character is called the newline character, one of the many escape sequences used for controlling the format of text. The `\n` character tells the compiler to insert a linefeed at that place in the text, just as if you pressed the <Return> key.

Any escape sequence can be recognized by the backslash (\) preceding it. The character following indicates which escape sequence should be executed. For example, the `n` signifies a linefeed: The computer starts the text after the `\n` at the beginning of the next line.

This newline (`\n`) escape sequence is important, since the `printf` function doesn't automatically add linefeeds after it displays text. You'll remember that when you ran the program in Chapter 3, the prompt appeared right after the text. Basically C writes all characters sequentially on the screen, even if they are written with different function calls. The user must program the function to advance the line.

The user is not obligated to place the escape sequences at the end of the character string. They can be placed between other "normal" characters or even at the beginning of the text. If desired, all three lines can be written within one `printf` function:

```
printf("Hello, I have a question!\nDo you believe in life
without electricity?\nNot me!\n");
```

This line is rather difficult to read. If you consistently place a newline (`\n`) character at the ends of strings, programs will be much easier to read.

The newline character is not the only method of formatting text. The `printf` is comparable to the `PRINT` command in BASIC or the `write` statement in Pascal. The `f` in `printf` indicates that the text can be output in a specified format. This function can process and display character strings and other values as specified by the programmer. Since these capabilities are quite extensive, we'll introduce them to the reader as needed.

4.4 Comments

The C language lets the programmer insert comments in the source code. This can be used to tell the reader what the source code is supposed to do. Comments have no effect on the speed or size of the compiled program. Comments start with the /* characters and end with the */ characters. The compiler skips over everything between the comment delimeters. Use comments liberally, since they never affect the final program and add to the readability of the source code.

```
void main()
{
/* This program outputs a text which starts */

printf("---Comments -- desired -- stop --\n");
/* here^ and  ends over at the opposite end^ */
}
```

Now, think of your first program from Chapter 3. If you added comments to tell a future reader exactly what this program did and when, the end result could look something like this:

```
/* Program from Chapter 3 of Schaun's book */
/* Amiga C for Beginners from Abacus (C) 1988 */
/* This program prints the words Hello, I am here" */
/* to the screen. Nothing else. */
#include <stdio.h> /* call standard i/o header file */
void main()        /* main function */
{                  /* start of function */
 printf ("Hello, I am here!"); /* show text on screen */
}                  /* end of function */
```

This is a very exaggerated example. It doesn't matter if you comment a program this clearly, though—the comments are ignored by the compiler.

4.5 Variables and Arithmetic

Our knowledge of the C language is still rather small. Right now you can display text on the screen and insert comments in source code.

scanf

It would be nice to be able to have a program accept input from the keyboard. The `scanf` function is the opposite of the `printf` function—it reads input instead of displaying output (more about `scanf` later).

Consider how a program would execute for questions and answers. First the `printf` function displays the question on the screen. Next the `scanf` function reads the user's input.

For example, pretend that a number will be input as a response. The following question asks for a number 1 or 2:

```
Are you well?
(1) = YES, (2) = NO
Number:
```

The number must be stored somewhere. C provides a series of variable types which can be used.

Variable types

Variables allow certain types of information to be stored in a computer. The variable type depends on the type of information you want stored. There are variables for characters; variables for strings; variables for different kinds and sizes of numbers; and variables for combinations of numbers.

4.5.1 Integers

The `int` variable type represents integer (whole) numbers. Variables of type `int` can accept whole numbers from (approximately) -32,768 to +32,767. The following program uses integer variables, and introduces a practical application of the `scanf` function:

```
/* scan1.c section 4.5 */
void main()
{
  int input;

      printf("Are you well?\n");
      printf("(1) = YES, (2) = NO\n\n");
```

```
    printf("Number: ");
    scanf("%d", &input);
    printf("\n\nYour input was: %d\n", input);
}
```

The first line of the `main` function, `int input;`, tells the compiler what to do with the variables. This line assigns a name to the variable. The program uses this variable name for access to the variable's contents. This variable name (`input`) indicates its purpose. Variable definition ends with a semicolon.

Format specification

The following lines are the `printf` functions. Next, the `scanf` function asks for the user's input in response to the text "`Number:`" Since a large number of data types exists, the input routine must be told what data can be expected. The format specification `%d` specifies the data type. Format specifications are similar to the escape sequences (the characters preceded by a backslash). The percent sign indicates a format specification; the `d` tells `scanf` that the value to be read in must be of type `int`. The name of the desired variable follows the string in quotes, separated from the string by a comma. An ampersand (&) precedes the variable name. This is important: If the system crashes the user should first examine the `scanf` function parameters to make sure they are correct.

If `scanf` contains the correct information, the user can enter the input in the running program. Enter a number and press the <Return> key. This value can be found in the variable `input`. The program can now use this number.

This program accepts the input then displays the variable using the `printf` function. The `printf` function must also be told what kind of data it must process. A format specification identical to the one which appears in `scanf` serves this purpose. Because of this, `printf` knows that an integer number will be passed, which must be placed at the location occupied by the format specification in the text. The variable name `input` follows the string in quotes, separated from the string by a comma. The input appears on the screen if the input was a 1 or 2 and not text (text is not allowed here). If you enter text, the input becomes a large random number (in Lattice C) even though this number never appeared in the input line. If you enter no input `scanf` scrolls the screen one line upward and waits for a new (more useful) input.

4.5.2 The `if` statement

It is rather boring to just let the computer repeat the input. It would be better to respond to the input. For example, have the computer respond to an entry of "1" with the message, "That is very good!", or an entry of "2" where the message could be, "I am sorry to hear that!" The

computer must be capable of comparing the value stored in `input` with other numbers. Depending on the results of this test, it must select one text or the other text. In programming this is known as an `if...then` construct. As in many other languages, C also has this capability (C doesn't require the `then`).

The variable is compared with 1. If the condition is found to be true, the statement following the `if` command executes. Here is an example:

```
if(input == 1)
  printf("That is very good!\n");
if(input == 2)
  printf("I am sorry to hear that!\n");
```

The conditions appear inside the parentheses so that the first `printf` statement only occurs when `input` equals 1. The same is true of the following `if` statement, with the difference that the text executes if `input` equals 2. Semicolons never follow the `if`.

The user who wants to experiment can try a semicolon after the second `if`. If you do so, the program acts as if the line `if(input == 2);` doesn't exist. After each input "I am sorry to hear that!" appears.

Adding the four lines above makes the program run properly, but it could be improved. Some users may be familiar with the BASIC statement:

```
IF A=1 THEN PRINT "That is very good!:
    ELSE PRINT "I am sorry to hear that!"
```

`else`

The C language also has an `else` statement which can only be used together with the `if` statement. The `else` statement executes only when the condition has not been met. This eliminates the second test.

```
if(input == 1)
  printf("That is very good!\n");
else
  printf("I am sorry to hear that!\n");
```

The second test is now improved. A modified version of the C program now appears as follows:

```
/* scan2.c 4.5.2 */
void main()
{
  int input;

      printf("Are you well?\n");
      printf("(1) = YES, (2) = NO\n\n");
      printf("Input Number: ");
      scanf("%d", &input);
      if(input == 1)
        printf("That is very good!\n");
      else
```

```
        printf("I am sorry to hear that!\n");
}
```

If several commands should be executed after the `if` (e.g., two `printf` functions), the second line cannot be written immediately after it. The following example wouldn't work:

```
if(input == 1)
  printf("That is very good!\n");
  printf("Hope you stay healthy!\n");/* Not like this! */
else
  printf("I am sorry to hear that!\n");
```

Statement block

Since only one line is executed after the `if`, something else must be done. Up to now only one statement has been described. It is time to describe a statement block. Placing several statements inside braces creates a statement block which is valid as a unit. This block can be placed following the `if` statement without problems:

```
if(input == 1)
  {
    printf("That is very good!\n"); /* Right! */
    printf("Hope you stay healthy!\n");
  }
else
  printf("I am sorry to hear that!\n");
```

The Amiga can do a lot of things, but what it does best is calculate (and fast). Let's start with addition. To add two values use the plus sign (+):

```
sum = number1 + number2;
```

The result for this example is stored in the variable `sum`. This variable must be defined at the beginning of the function, just like all variables. Using type `int`, the following definition results:

```
int sum
int number1
int number2
```

C is a language for lazy people. Most everything can be changed to abbreviations to cut down on keyboard use. Those who want to become good C programmers should use this capability. These three variables are defined as the same data type. You only need to enter `int` once; all integer variables can be listed following the single `int`:

```
int sum, number1, number2;
```

All variables are separated by commas and the list is terminated with a semicolon.

With this information it's possible to write a program that adds two numbers. The `scanf` function permits data input, but this time two

numbers will be read in. This will not be done with two separate function calls (this would also be possible), but a second format specification is written into the string of the `scanf` call. It contains `%d%d` (notice no spaces) which reads the second parameter. And now the listing:

```
/* scan3.c 4.5.2 */
void main()
{
  int sum, number1, number2;

  printf("Please input two numbers!\n");
  scanf("%d%d", &number1, &number2);
  sum = number1 + number2;
  printf("%d + %d = %d\n", number1, number2, sum);
}
```

The user familiar with other languages such as BASIC can compare the listings to other language implementations of the same program. The last `printf` function with the three format specifications can appear confusing. The following is a BASIC equivalent:

```
PRINT "Please input two numbers!"
 INPUT N1,N2
 SUM = N1 + N2
PRINT N1;" +";N2;" =";SUM
```

4.5.3 Calculating with C

The reader can guess what would have to be changed to perform multiplication instead of addition. The plus sign is replaced by an asterisk (*). A hyphen (-) is used for subtraction and a slash (/) performs division.

Calculations may be performed with constant values as well as variables. Both constants and variables can be mixed. Some examples follow to illustrate. It is assumed that all variables are defined and contain meaningful values:

```
result = number * 4;
sum = var + 2 + var2 + 3 + var4;
result = 4 * 5 - 7 / var;
value = 2 * (number - 7);
result = 4 + 5 * 3 - 2;
counter = counter + 1;
```

Evaluating a formula is as simple as entering it into a pocket calculator. C recognizes the laws of mathematical precedence:

```
result = 4 + 5 * 3 - 2;
```

The result is 17, not 25. The product of `5 * 3` is calculated first, after which 4 is added and 2 subtracted.

The `counter = counter + 1;` example above deserves special attention. This equation is unsolvable for the normal person, but no problem for the computer. It takes the content of the variable `counter`, adds one to it and stores the result in `counter`. This operation increments the content of the variable by one for every call.

Let's write a comprehensive program for performing math equations. The assumptions are that all basic four mathematical functions are performed with two variables. A `scanf` function reads the numbers. Then the numbers are tested with several `if` statements to determine which mathematical function should be performed. The result is calculated accordingly. If an invalid code is input, a message appears.

Here is the finished version of such a program:

```
/* math1.c 4.5.3 */
void main()
{
  int number1, number2, result, operator, error;

  printf("Please input two numbers!\n");
  scanf("%d%d", &number1, &number2);
  printf("And now the code for the operation!\n");
  printf("1=Add, 2=Subtract, 3=Multiply, 4=Divide\n");
  scanf("%d", &operator);
  error = 1;
  if(operator == 1) /* Addition */
    {
     result = number1 + number2;
     error = 0;
    }
  if(operator == 2) /* Subtraction */
    {
     result = number1 - number2;
     error = 0;
    }
  if(operator == 3) /* Multiplication */
    {
     result = number1 * number2;
     error = 0;
    }
  if(operator == 4) /* Division */
    {
     result = number1 / number2;
     error = 0;
    }
  if(error == 1) /* None of the above conditions  */
  printf("Wrong Code! Input only numbers 1 - 4!\n");
else
```

```
  printf("The result is %d\n", result);
}
```

The program is very easy to read. After all values have been entered, the variable `error` is assigned a value of 1. During every operation that follows, be it addition, subtraction, etc., the `error` variable is set to zero. This makes it possible to determine whether one of the four operations was performed. If this was not true, the value in `operator` is illegal.

Before the output, the result is tested to determine if it was calculated. This can be seen in the variable `error`.

Test this program thoroughly with various values. Please note that the integer variables are only permitted to store values between +32,767 and -32,768. Furthermore, division by 0 should be avoided. This would cause a system crash and a Guru Meditation.

4.5.4 Floating Point Numbers

Perhaps you have noticed something else. Try dividing 9 by 2. The result displayed by the computer is 4, which is incorrect (4.5 would be right). Isn't this expensive computer capable of performing correct division?

The error can be traced to the variable type. The `int` variable type is only capable of processing whole numbers between ±32,000. The value 4.5 is a floating point number, not a whole number. If during a division a remainder (the fraction after the decimal point) occurs, it is ignored. This does not mean that the division 9/2 cannot be performed on the Amiga computer. The only thing required is that the variable type can be capable of storing floating point numbers. No problem since C is equipped for this.

`float`

To convert the current program for this new data type, the variables `number1`, `number2` and `result` must be changed. This process consists only of replacing `int` with `float`. The first lines appear as follows:

```
main()
{
  float number1, number2, result;
  int operator, error;
```

This alone is not sufficient since `scanf` and `printf` use the format specification `%d` which expects an integer value. This is no longer the case. The `%d` must be replaced with a `%f`. The `f` means a floating point

value is passed, just like the `d` is used for integer values. The first `scanf` function now appears as follows:

```
scanf("%f%f, &number1, &number2);
```

The last `printf` function must also be changed. The variable `result` is now a floating point value. The `d` is replaced with the `f`. Newly compiled and linked, this version makes error free computations of floating point numbers possible. Numbers stored as type `float` are practically unlimited in size. Millions, and even billions and billions, can be calculated. Here is the complete program:

```
/* math2.c 4.5.4 */
void main()
{
  float number1, number2, result;
  int operator, error;

  printf("Please input two numbers!\n");
  scanf("%f%f", &number1, &number2);
  printf("And now the code for the operation!\n");
  printf("1=Add, 2=Subtract, 3=Multiply, 4=Divide\n");
  scanf("%d", &operator);
  error = 1;
  if(operator == 1) /* Addition */
    {
     result = number1 + number2;
     error = 0;
    }
  if(operator == 2) /* Subtraction */
    {
     result = number1 - number2;
     error = 0;
    }
  if(operator == 3) /* Multiplication */
    {
     result = number1 * number2;
     error = 0;
    }
  if(operator == 4) /* Division */
    {
     result = number1 / number2;
     error = 0;
    }
  if(error == 1) /* None of the above conditions  */
  printf("Wrong Code! Input only codes 1-4!\n");
else
  printf("The result is %f\n", result);
}
```

Lattice

The library for mathematical functions and floating point numbers must be linked with the standard library. Example:

```
lc -Lm  math2
```

Aztec

If you work with the Aztec C compiler, the library for mathematical functions and floating point numbers must be linked with the standard library `c.lib`.

Example:

```
cc +L math2.c
ln math2.o -lm -lc
```

4.5.5 Characters and Character Strings

`char`

Besides the `int` and `float` variable types which accept numbers, you need another category of variables to store characters. It would be better if the program above could accept a plus sign instead of the number 1 to indicate addition. The data type `char` allows variables to be defined which can accept characters.

The syntax for definition of a `char` variable is exactly as described in the `float` and `int` variables:

```
char character;
```

This type of variable has its own format specification for the `printf` and `scanf` functions. A `c` is used for the type `char`. To give the calculation program a few extras, the operator is entered as a character. This means that instead of entering a number as you had to before, you can enter a math operator instead.

Another improvement can be made at this point. The format for entering equations should be similar to that of a pocket calculator (i.e., first number, operator, second number). You should be able to press the <Return> or <Enter> key instead of the <=> key. Since the `scanf` function is so flexible, the following change is sufficient to make this possible:

```
scanf("%f%c%f", &number1, &operator, &number2);
```

The `%c` between the two format specifications indicates character input.

The tests which formerly checked the code now have to test the characters for the operators. Nothing easier than that! Only the characters must be placed in apostrophes (single quotes):

```
if(operator == '+')
  ...
```

After all the small changes, compare this version to the final program below in which some other cosmetic changes were made. The reader should now be able to understand the additions made.

During the input a small item has changed. Until now the <Return> key had to be pressed (but not required) after inputting each number. Now the entire input must be in one line. The reason for this is the fact that a single character is read in with `%c`. This could be a <Return> or a space. For this reason the first number is followed immediately by the operator after which the <Return> key may be pressed, if desired. Finally the second number appears as in this line:

```
15.500000 * 12.500000 = 193.750000
```

Here is the final version of this program:

```
/* math3.c 4.5.5*/
void main()
{
  float number1, number2, result;
  char operator;
  int error;

  printf("Input Format: Number, Operator, Number (no
spaces)!\n");
  scanf("%f%c%f", &number1, &operator, &number2);
  error = 1;
  if(operator == '+') /* Addition */
        {
     result = number1 + number2;
     error = 0;
    }
  if(operator == '-') /* Subtraction */
        {
     result = number1 - number2;
     error = 0;
    }
  if(operator == '*') /* Multiplication */
        {
     result = number1 * number2;
     error = 0;
   }
  if(operator == '/') /* Division */
        {
     result = number1 / number2;
     error = 0;
   }
  if(error == 1) /* None of the conditions above
satisfied? */
  printf("Wrong Operator %c!\n", operator);
else
  printf("%f %c %f = %f\n", number1, operator, number2,
result);
}
```

Lattice

The library for mathematical functions and floating point numbers must be linked to the standard library. Example:

```
lc -Lm  math3
```

Aztec

If you work with the Aztec C compiler, the library for mathematical functions and floating point numbers must be linked with the standard library `c.lib`. For example:

```
cc +L math3.c
ln math3.o -lm -lc
```

5.
Loops

5. Loops

The programs presented up until now execute straight from beginning to end. An `if` statement may skip over some spots, but we haven't jumped to earlier statements. Loops branch to earlier sections of the program.

5.1 `while` Loops

The `while` statement is followed by two parentheses which surround the desired arguments. An example makes this clear:

```
void main()
{
  int counter;

  counter = 15;
  while(counter > 0)
      {
      printf("Counter is %d\n", counter);
      counter = counter - 1;
      }
}
```

The block which appears after the `while` statement executes until the conditions inside the parentheses are true. In the beginning, the variable `counter` is set to 15. While the condition `counter > 0` has been met, the following block is executed which outputs the current value of `counter` and then decrements it by one. In this case the `printf` function is called 15 times until `counter` has been reduced to 0. The conditional statements can be formed by using tests for equality (==), greater than (>), less than (<), greater than or equal to (>=), or less than or equal (<=). The comparison here is for `counter` to be greater than 0.

The comparison operators for C are similar to those found in most programming languages.

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	unequal

Instead of

```
while(counter > 0)
```

the following could be written:

```
while(counter >= 1)
```

The latter is preferred since the limit is explicitly provided. To construct a loop which counts up to the value 100, it is recommended to use this number in the test.

```
while(counter <= 100)
```

In the comparison operators <= and >=, the equal sign (=) always appears at the end.

Factorials

A practical example is the calculation of a factorial number through constant multiplication. A factorial in mathematics is the product of all whole numbers up to a set value. The factorial of 4 therefore is:

```
4! = 1 * 2 * 3 * 4 = 24
```

Here is our example program:

```
/* factorial.c 5.1 */
void main()
{
  int num, i;
  float factorial;

  printf("Please input a number: ");
  scanf("%d", &num);
  i = num;
  factorial = 1;    /*Initialize */
  while(i >= 1)
     {
     factorial = factorial * i;
     i = i - 1;
     }
  printf("%d! = %f\n", num, factorial);
}
```

Now for a few hints in helping you compile this program with your compiler.

Lattice

The library for mathematical functions and floating point numbers must be linked with the standard library. Example:

```
lc -Lm factorial
```

Aztec

If you work with the Aztec C compiler, the library for mathematical functions and floating point numbers must be linked with the standard library `c.lib`. Example:

```
cc +L factorial.c
ln factorial.o -lm -lc
```

5.2 for Loops

Another loop can be constructed using the `for` statement. In BASIC the command is used as follows:

```
FOR I = 0 to 100 STEP 2
.....
NEXT I
```

In C this appears as follows:

```
for(i = 0; i <=100; i = i + 2)
.....
```

C doesn't require a `NEXT` as in BASIC, since only the statement block following the loop header is executed. Within the parentheses are some interesting items. There are three individual statements separated from each other by semicolons. A semicolon does not follow the last entry. The loop body ends here with the closing parenthesis. The first entry `i = 0` assigns a starting value to the variable which is modified within the loop. The statement `i <= 100` represents the ending condition. Until it is satisfied, the loop executes. The last part of the loop body increments or decrements the control variables which were previously initialized with a starting value.

5.3 do while Loops

The last type of loop is the do while loop. The reader has already read about the while loop; this loop is quite similar. Please compare the two program sections below:

```
/* first program section */
while(i > 0)
  {
     i = i - 1;
     printf("i is %d\n", i);
  }

/* second program section */
do
   {
     i = i - 1;
     printf("i is %d\n", i);
   } while(i > 0);
```

The do begins the do while loop, and the while ends the loop. This leads to a small but significant difference in program execution. In the first example the program checks if variable i is still greater than 0 and then executes the loop only if the conditions are met. In the second example the test is executed only after the loop has already been executed once. If i contains the value 0, the while loop is skipped, unlike the do while loop which executes at least once.

Please notice the semicolon which must follow the while. It is often forgotten since normal while loops don't use semicolons.

If this material is not clear without further example programs, the user is encouraged to write some short programs (e.g., which output the values of the variables used).

5.3.1 More Error Checking

The next section deals with error detection. Up to now it was difficult to make mistakes, except for errors in typing. When the program suddenly reports errors, there's no need to panic. Study the messages the C compiler returns. You may have to do a little thinking to detect cleverly hidden errors.

Find the errors

Below is a new program. Based on what you know so far you should be able to determine where the errors are hidden, and which lines could cause problems. The listing contains errors which result in a long series of error messages. Try to find the hidden errors on your own first. Fix these errors, then try compiling the source code to see what you missed. We've included the solution directly after the listing. The program should add all numbers from 1 to 100 and display the subtotals and the final total on the screen.

```
main();
{
  printf("I add all numbers from 1 to 100/n");

  i = 1;
  do
     printf("Subtotal for %d. value: %d/n", i, sum);
     sum = sum + 1;
     i=i+1;
  while(i < 100)
  printf("Sum of all numbers to 100 is %d/n", sum);
}
```

Did you find all the errors? You should have found most of them, since the program is almost completely wrong! Even if you found no errors, you can follow the remaining material without problems.

Let's start with the first line which contains an error (of course). The semicolon following `main` shouldn't be there. The missing `void` only results in a warning, not an error. The next line with the brace is correct (an exception in this program). The compiler accepts the first `printf` function without problems. It doesn't contain a syntax error. The line would even be right if you wanted to display the slash (/) and an `n`. The slash (/) should have been a backslash (\). To be consistent, this error occurred in all the `printf` functions in this program.

The assignment `i = 1;` is correct. The `do while` loop, which should execute the following three lines, has no braces (the braces make the three lines into a statement block). The semicolon is also missing after the `while (1<100)` line.

Within the loop, the values for `i` and `sum` should be displayed. Except for the error with the escape sequence `\n` everything is correct here. Finally the program increments the contents of `sum` and `i` are incremented. Trouble is, the `sum` variable was never defined. The C compiler doesn't know what is meant by `i` and `sum`. A line must be added before the `printf` function.

```
int sum, i;
```

Before or after `i = 1;`, a `sum = 0;` must be added. After these changes the C compiler is happy but the program will not display the right output. There are still two logical errors in the program. The first

occurs as the subtotal is displayed. The value of sum is displayed before it has been calculated. The line sum = sum + i; must be placed before the line with:

```
printf("Subtotal  for %d. value: %d\n", i, sum);
```

The while test remains which terminates the program after the number 99. The change is simple:

```
} while(i <= 100);
```

The error free version of the program appears below:

```
/* errorfree.c 5.3.1 */
void main()
{
  int sum, i;
  printf("I add all numbers from 1 to 100\n");

  sum = 0;
  i = 1;
  do
     {
       sum = sum + i;
       printf("Subtotal for %d. value: %d\n", i, sum);
       i = i + 1;
     } while(i <= 100)
  printf("Sum of all numbers to 100 is %d\n", sum);
     }
```

5.4 AND and OR

Up to this point, only one exit condition can be checked in your loop. This changes with the introduction of the && and || operators. The <|> key can be found on the right side of the keyboard above the <Return> key. The && represents the logical AND and the || the logical OR. Why these operators are called logical will be revealed later since other logical operators also exist in C. From BASIC the commands AND and OR are familiar and they are the same as the operators in C.

AND

Connecting two conditions with AND:

```
while(i <= 10 && i >= 5)
      ...
```

The loop is now executed when

- i is less than or equal to 10.
- i is greater than or equal to 5.

If one of the two criteria is not met (e.g. i = 4), the entire condition is false and therefore not satisfied. Only if both tests are true can the loop be executed.

OR

OR is used as it is in daily conversation. If one of the two conditions is true, the entire expression is true. The next example assumes that a character variable should be tested for a certain content. Since the logical connections can be used with other conditional tests, they can be used together with if:

```
if (operator == '+' || operator == '-' || operator == '*'
   || operator == '/')
   printf("The operator is valid!\n");
```

Four tests were made, of which only one must be true. If several tests can be positive, this is no problem since only one true condition is sufficient. That the if statement could be written in two lines should be nothing new. Remember that the formatting of the C listing is of no interest to the C compiler.

Negation

There is another operator to be discussed. This is the negation operator !, mentioned as a part of the inequality operator !=. With this character all tests and returns can be made into the opposite. If a test should be made to determine if a character is not an arithmetic operator, the following test can be devised:

```
if(  !( operator == '+' || operator == '.' || operator ==
'*' || operator == '/'))
printf("Not a valid Operator!\n");
```

All tests are made within the parentheses. If the expression inside the parentheses is true, a valid character is present, the negation operator goes into action. It simply reverses the matter. From the true test it makes a false one so that the `printf` command is not executed. This is similar to the false test result within the parentheses, when none of the signs + - * / are stored in the variable. In this case the ! operator makes it a true test. That is the same procedure as inserting a `not` into a sentence. In everyday English double negatives can be used in one sentence, but not many people will understand it.

6.
Strings

6. Strings

You've entered strings and displayed them on the screen in previous chapters. What else can you do with them? The following code shows string variable definition:

```
char name[number_of_fields];
```

Strings consist of groups of individual characters of type `char`. The above variable definition tells the compiler the maximum number of characters the string can have. If you want to process a single character belonging to the string, you can't just call the variable—you'll get the entire string. In addition, you must know the exact location in the string at which you can find the specific character. You'd enter the number of the character in brackets, just as you did in the definition.

Let's take the first character in a string. This first character appears in the first position of the string, and is assigned position 0 (computers always start counting with 0). All locations then shift by one. The second character can be reached using the value 2, due to the index, which acts as a position indicator. Every position contains a character. All characters are arranged sequentially in a large or small string.

6.1 Backtracking

Let's write a program which displays the text backwards on the screen. Before starting, you must assume that a string can be any length. The string will always end with the value 0 (null). The last character of the string must be processed first if you want the text displayed backwards. The program needs a small `for` loop to find the last character of the string (0):

```
...
char input[81];
int index;

for(index = 0, input[index] != 0; index = index + 1)
   ;
...
```

The loop body (the statements executed during every pass through the loop) is empty. A single semicolon follows the `for` loop. Since a block of statements follow every loop, this semicolon ends a block that

does nothing. This is the empty statement. The data in parentheses perform all the required operations. First the index is set to 0. Then the test follows which determines whether a character is not equal to 0. If the condition is satisfied, the index is incremented by one. The last element of the string contains a zero and the loop ends. The result in index is the length of the string. The last character of the string is located one position before the null value. Therefore the index variable must be reduced by one before being used. The index counts down to zero one step at a time and the program displays a character at every step.

```
do
  {
     index = index - 1;
     printf("%c", input[index];
} while(index > 0);
```

You now have the information you need to write the entire program. One other item before you enter and compile this program: If you have the Aztec C compiler, this program will not compile using the +L (longwords) option. Omit this option when compiling this program with Aztec C. Here's the source code:

```
/* backwards.c 6.1 */
void main()
{
  char input[81];
  int index;

  printf("Please input some text!\n");
  scanf("%s", input); /* Strings do not require & */

  for(index = 0; input[index] !=0; index = index + 1)
    ;        /* Search for End mark! */

  printf("Your input >%s< has %d characters\n", input,
index);

do

  {
     index = index - 1;
     printf("%c", input[index]);
  } while(index > 0);

printf("\n\n");  /*Blank line before Prompt */
}
```

Enter a string of characters, but do not include any spaces. The `for` loop calculated the length of the input string and then the `do while` loop printed it out backwards. Former BASIC programmers may remember the `LEN` function. In C, the user can easily create a function or routine to do this.

7.
Calculating in C

7. Calculating in C

You have already seen how fast the Amiga computes; addition (+), subtraction (-), multiplication (*) and division (/) are familiar to you.

Modulo operator

The modulo operator (%) performs another mathematical operation—modulo division, which calculates the remainder of an integer division. The result is assigned to a variable using the equal sign, where the variable must be to the left of the assignment operator. The general format is as follows:

```
Variable = operand1 <> operand2
(<> represents the operator)
```

First the program calculates the expression to the right of the equal sign and places the result in the variable to the left of the equal sign. Because of this, statements such as the following are possible:

```
variable = variable + 1;
```

This expression is impossible in normal math, but poses no problem for the computer. The computer reads the variable content, adds one and stores the result in the same variable. Combinations of math operations are possible as the following examples show:

```
number = 3 * 32
number = 2 + 6 * 7
number = 5 * (180 / 3 + 9) * (5 - 2)
number = number - 1
number = number % 2;
```

Precedence

Division and multiplication have precedence over addition and subtraction. This rule is also observed by the C compiler. Therefore, the expression 2 + 6 * 7 needs no parentheses to achieve the correct result. The modulo operator has precedence equal to division, and therefore precedence over subtraction and addition. Example:

```
5 % 3 = 2, since 5 / 3 = 1 and remainder is 2.
```

Simple calculations don't need to use a variable. The following program will illustrate:

```
void main()
  {
    int number;
    number = 3 * 12;
    printf("Result: %d\n", number);
  }
```

A variable does not have to be used in this program since the `%d` characters tell `printf` that it can expect an integer value. The term `3 * 12` can be passed directly to the function as a parameter. The calculation of the result occurs before the value is passed so no variable is required. The following program is faster and shorter:

```
main()
  {
     printf("Result: %d\n", 3 * 12);
  }
```

An integer number can contain only a whole number, so the statement `number = 3 / 2` places a value of 1 in the variable `number`. The correct result would have been 1.5, but the result will be rounded to the next whole number.

Note: The number -2.25 is rounded to -2 and not -3 since -2 is larger than -3. Lattice C rounds numbers toward 0, but this can differ with other C compilers. Only a test run helps to explain what happens with -5 / 2, in which either -2 (toward zero) or -3 (rounded) appears as a result.

8.
Variables

8. Variables

The earlier chapters used variables. These are areas of memory used for storing mathematical results, as well as different kinds of data.

Variables are subject to certain rules and regulations. They must be assigned specific data types and unique names. As you'll see in this chapter, there are many types, and the names can be almost anything you want them to be.

8.1 Variable Names

The names given to variables must follow some rules. The following is a list which describes these rules:

1. The first character must be a letter (the underscore character [_] counts as a letter); after the first character, any legal character can be used.

2. Characters within variable names can be letters, numbers or underscore characters (the shifted minus sign).

3. No control characters or foreign characters allowed.

4. Variable names can be of any length, but many C language compilers use only the first eight characters of the variable name (the Lattice C compiler permits up to 30 valid characters, Aztec up to 31 characters).

5. Reserved C keywords may not be used as variable names.

6. Variable names are case sensitive (i.e., the compiler sees a difference between upper and lowercase).

Some examples are shown to make these rules more understandable. Some are correct and some aren't. Can you find the bad names?

a) `Number_1`
b) `2_pi`
c) `first-var`
d) `Book_no_1`
e) `Book_no_2`
f) `int`
g) `_flag`
h) `int_valu`
i) `number_1`
j) `secret_Password`

The following variable names are correct: a), d), e), g), h), i) and j). It should be noted that a) and i) are different variables since upper and lowercase letters are differentiated. d) and e) may refer to the same variable on some C compilers, since the names are the same for the first eight letters. Errors would occur on some C compilers since these variable names are longer than eight characters: c), d), e) and j). h) uses a C keyword as a variable name, but this is permitted since the rest of it doesn't match the keyword.

Look at examples b), c) and f). The variable in b) starts with a number (not allowed). A hyphen appears in c) (the hyphen is considered a special character). Finally f) uses a variable name which is a reserved keyword of C.

The following list shows the reserved words used in C:

```
auto        enum        short
break       extern      sizeof
case        float       static
char        for         struct
continue    goto        switch
default     if          typedef
do          int         union
double      long        unsigned
else        register    void
entry       return      while
```

8.2 Data Types

Until now three data types have been described: `int`, `float` and `char` (strings).

`int`

Integer values are type `int`. The 16-bit `int` type represents whole numbers between -32,768 and 32,767. The `int` type works well for general use. The Amiga libraries use 32-bit integers, but for portability of your source code to other computers you may want to use 16-bit integers.

`float`

Floating point numbers are assigned the data type `float`. A `float` variable can store extremely large or small numbers, and numbers with decimal places. In this type of variable the values can be presented in scientific notation. Very small numbers such as 0.00000015 can be written as 15E-7 (the use of E is an abbreviation). 15E-7 is scientific notation for:

$$15 * 10^{-7}$$

Numbers with as many as 15 places can be written. The "e" which separates the exponent (here -7) from the mantissa (in this case 15), can be written in upper or lowercase letters. In both cases that compiler will translate it without problems.

Even the `float` variable type has its limitations. The largest number permitted is 10^{38}. Any number less than 10^{-38} converts to a 0. The value 10^{-40} when written out is a number which has a decimal point, 39 zeros and a one, in that order. The computer views it as 0. Floating point variables remain accurate up to seven decimal places. Try the `math3.c` calculation program from Section 4.5.5; enter the number 16.8. The program converts the number to 16.799999.

`double`

If you need more accuracy, use the `double` variable type. Variables of type `double` are about twice as accurate (11-14 decimal places). However, `double` variables require more memory.

There are times when `float` and `double` variable types don't have the accuracy of integer values. On the other hand, rounding numbers off can cause incorrect results in multiple calculations. The result becomes more inaccurate with every additional operation.

Avoid comparing a fixed value during a test. For example:

```
if(value == 1.0)     /* Not like this */
```

It would be better to test if it is larger or smaller so that the value tested is not skipped through a rounding error. Otherwise an infinite loop could result.

Special conditions must be considered when using floating point numbers. To determine if a constant without fractions following the decimal point (for example 2) is a floating point number, another digit must be added after the decimal point. This error occurs during the `printf` call in the following example:

```
printf("Result of 2 / 3 = %f.\n", 2/3);
```

The example computes 2/3 as an integer value and passes 0 as a result. The function waits for a floating point number which was indicated by `%f`. The example below is the correct version:

```
printf("Result of 2 / 3 = %f.\n", 2.0/3.0);
```

Note: You may crash the system as well as get the wrong answer with the example above. If you want to try the two examples above, save any important data you might have on the RAM disk to a floppy disk before continuing.

`char` Other data types can be derived from the basic types `int` and `float`. For example, the type `char` which can accept a character is really a variable for whole numbers between -128 and 127. This small relative of `int` represents the ASCII values of the characters.

`long` The `long` type is another type derived from `int`. Long accepts integers between -2,147,483,648 and 2,147,483,647. If you must define constants as long values, place an `l` or `L` behind the floating point number instead of .0. For example:

```
1L
```

`short` The C keywords `unsigned` and `short` can be used as adjectives to the basic types. These specify integer values. The combinations of `unsigned` and `short` cannot be used with `float` values. `unsigned` defines an integer number which has no sign; `short` accepts only 16-bit numbers.

If the indication of `int` or `float` is missing, C defaults to `int`. The following combinations are valid:

```
unsigned = unsigned int
short = short int = (Compiler dependent) char
long = long int
unsigned long int
long float = double
```

The advantage of `unsigned` is limited to positive values, and extends the limit of the normal type. For example, `unsigned int` accepts numbers between 0 and 65,535. The normal `int` type only permits positive values up to 32,767. Unsigned numbers also allow operations which cannot be performed with other types. More on this later.

The value assignment to `char` variables proceeds in the following manner, as in `int` values:

```
char character;
character = 65;
```

`char` stores characters. This is done with the following assignment which leads to the same result as the line above.

```
character = 'a';
```

The combination of integer values with the attributes `short`, `long` and `unsigned` return different results from one compiler to another. For this reason no general value or memory requirement can be provided. The following relationship exists between the length of variables used by all C compilers:

```
char <= short <= int <= long
```

The table below describes the length of the different numeric variable types:

	Lattice	Aztec
char	1 byte	1 byte
short	2 bytes	2 bytes
int	4 bytes	2 bytes
long	4 bytes	4 bytes

8.3 Type Conversion

Type conversions sometimes become necessary during computation because of the use of various data types. The following rules govern type conversion:

1. `char` and `short` always convert to `int`; `float` always converts to `double`.

2. If after these conversions one of the operators should have the type `double`, the second operand and the result also convert to `double`.

3. If a data type is now `long`, all participating values also convert to `long`.

4. If an `unsigned` value exists among the operands, all values convert to `unsigned`.

8.4 The `cast` Operator

Constants, function values and variables can be converted into a specific data type. The parameter to be converted is placed in parentheses and is preceded by a data type in parentheses. This is the *cast operator*. The parentheses are not always required, but are recommended because of the high precedence of the type conversion. For example:

```
long number;
number = 123 / (long)('a' / 1.5);
```

In a general format, the expression is:

```
(type) Parameter
```

Better format:

```
(type)(Parameter)
```

Any data type can be substituted for the word `type`.

9.
printf and scanf

9. printf and scanf

The most powerful output function in C is `printf`. You've seen a little of what it can do with screen output using examples printed earlier in this book.

The `scanf` function gives the user the option of input to the computer. You have had a chance to work with this function as well.

Both `scanf` and `printf` use a number of format specifications and escape sequences for controlling the format and type of input and output.

9.1 More Escape Sequences

You'll remember reading about the `\n` escape sequence in Chapter 3. The following list shows other escape sequences, which you'll find useful for controlling text output.

`\t`	places output at the next tab stop (all 8 positions)
`\b`	moves the current write position one place to the left (backspace)
`\r`	inserts carriage return at first position of the current line
`\n`	inserts carriage return and linefeed
`\f`	inserts form feed
`\\`	prints the backslash character (\)
`\"`	prints quotation mark within the string
`\'`	prints apostrophe within the string
`\nn`	prints any character with the octal value `nn`

Note: An escape sequence uses two characters in the text, but represents only one character. Keep this in mind when calculating memory usage.

The following program uses the `\t` escape sequence for tab stops:

```
main()
  {
     printf(An\tExample,\ttwo\tTabs\tspacing");
     printf("\tspacing\tText!\n");
  }
```

For our next assignment we wish to display the following text:

```
Use the control characters : "\n", "\t"!
```

It is not possible simply to place the text in quotation marks since they already occur in the text. Escape sequences are necessary. The program prints the quotation mark using the \" escape sequence, the backslash with \\. The necessary `printf` call appears as follows:

```
printf("Use the Control Characters: \"\\n\",\"\\t\"!");
```

Here is an example program:

```
main()
  {
     printf("Small ");
     printf("\"T e s t  p r o g r a m");
     printf("\"\n\nWhere\nis the\ntext now?\n");
     printf("\tEverything OK?\n");
  }
```

The output is:

```
Small "T e s t  p r o g r a m"
Where
is the
text now?
     Everything OK?
```

9.2 Format Specification

Strings can accommodate format specifications as well as text and escape sequences. Every format specification has a corresponding variable attached to the string and separated from the string by a comma. The format specifications always start with a percent character (`%`) and can be used in the `printf` and `scanf` functions.

A difference from the format string of the `printf` function is important: The `scanf` function reads in data. For this reason the escape sequences `\n` (linefeed), `\t` (tab) and space divide the input into separate fields.

Here is a table with format specifications for `printf` and `scanf`:

Format specification	Data type
`%c`	`char` (one character)
`%d`	integer value
`%s`	string
`%f`	`float` and `double` number (for `printf` output with format [-]xxx.xxxxxx)
`%o`	integer value as octal number (base 8)
`%x`	integer value as hex number (base 16)
`%u`	unsigned integer value (`printf` only)
`%e`	`float` or `double` output (`printf` only) in scientific notation [-]x.xxxxxxE[+-]xx: affects `scanf` as in `%f`
`%g`	shortest form of `%e` and `%f` (`printf` only)
`%h`	short (`scanf` only)
`%%`	represents the `%` character (`printf` only)

Additions

The integer elements `d`, `u`, `o` and `x` can be preceded by the letter `l` to indicate that long values instead of integers are used. Long values are integers of double length. An `l` preceding floating point numbers containing `e`, `f` and `g` indicates that double values are expected. The field width of the input or output of a field can also be indicated with a format element. After the percent sign, the size of the individual field can be indicated. If the first character of this number is a minus sign, the text is left justified. Spaces fill the remaining positions in the field.

Without indication of the field width, the standard setting for `%f` in the `printf` function is `%.6f`. The output therefore always has six decimal places and any field size.

`printf` and elements for `float` values:

```
%<min>.<fraction>F
```

<min> indicates the minimum width of the output field. <fraction> is the maximum number of numbers after the decimal point. F is one of the format specifications e, f or g. The indication of 0 for fractional positions truncates all numbers after the decimal point (e.g., %.0f). For example:

```
printf("Number %5.2lf\n", 12.345);
```

creates the output:

```
Number 12.35
```

In this case the program expects a double number (lf) at least 5 characters wide, but with only 2 decimal places. Since rounding is performed to the second number after the decimal point, the number 5 appears at the last position. If less numbers are available than the number indicated for positions after the decimal point, zeros are attached.

printf and elements for integers:

```
%<Min>F
```

<Min> indicates the minimum width of the output field. F represents one of the format instructions d, u, o or x. Example:

```
printf(">%4d<", 12);
```

Output:

```
> 12<
```

printf and %s:

```
%<Min><real>s
```

<Min> indicates the minimum width of the output field, while <real> describes the actual number of characters displayed. The following examples show the effects on the string "Sampletext":

Format Specification	Output
>%6s<	>Sampletext<
>%-6s<	>Sampletext<
>%12s<	> Sampletext<
>%-12s<	>Sampletext <
>%12.6s<	> Sample<
>%-12.6s<	>Sample <
>%.6s<	>Sample<

The scanf function is much simpler. Only one number exists which indicates the maximum input length possible. As soon as a character no longer fits into the format of a data type, or a control or blank character appears, the input for the current field ends. This means that during

input only characters representing an integer number are used. If other characters are input, the integer number input ends. In addition '*' can be used which precedes the format instruction for data type and suppresses the assignment. The field is simply skipped in this case.

```
int i;
float f;
char string[50];
scanf(%3d %f %*d %s", &i, &f, string);
Input: 1234567.89 12345all clear?
```

Value assignment

`i` contains 123, since the field should have 3 places, and only numbers can appear. The value 4567.89 is in `f`, because the space after "9" prevents additional reading of input. The same happens after the storage of "all" in the `string[]`. The number sequence "12345", which normally is assigned to an integer value, was skipped because of the asterisk. This means that with the `scanf` no spaces can be read. This makes `scanf` less than ideal for string input.

If the reader can't remember all of this material, don't worry. It is used intensively during the course of the format specifications.

9.3 Octal and Hexadecimal

Two number systems are used often in C. To discuss these systems, we'll start by looking at the decimal system. Let's take a decimal number and dissect it into its component parts:

```
5279
  =     5,000       + 200       + 70         + 9
  =     5 * 1,000   + 2 * 100   + 7 * 10     + 9 * 1
  =     5 * 10^3    + 2 * 10^2  + 7 * 10^1   + 9 * 10^0
```

This makes the origin of the term decimal = 10 in our number system clear. Every number position has a certain value. There are ones, tens, hundreds, etc. The value of these positions is multiplied with the number at that location. For example 7 * 10. The factors 1, 10, 100, 1,000 can be traced again to the base 10. The exponent in base 10 depends on the position of the digit in the number. The first position corresponds to exponent 0, the second exponent 1, the third 2 and so on.

Numbers between 0 to 9 can be used, which makes ten different numbers available. This is the reason the system is called base 10.

Octal system

If you used eight different numbers (0-7) instead of ten, the base in the calculations would be 8. This base 8 system is better known as the octal system. The following example shows the process of calculation a number in the octal system. To differentiate the different number systems, the base number appears in subscript or in parentheses following the number:

```
6204
   8
  = 6 * 8^3       + 2 * 8^2    + 0 * 8^1    + 4 * 8^0
  = 6 * 512       + 2 * 64     + 0 * 8      + 4 * 1
  = 3,072         + 128        + 0          + 4
  = 3,204
       10
```

This brings us back to the format specifications. To write a variable in octal on the screen, use %o.

```
printf("3204 dec. = %o octal\n",3204);
```

Hexadecimal

The format specification `%x` converts a number into the hexadecimal system. As the name suggests, hexadecimal is base 16. This produces a small problem. The decimal system uses ten numbers (0-9), but hexadecimal notation needs 16. Hex notation uses the first six letters of the alphabet as the top six numbers. The letter A represents the number 10; B has the value 11; C 12; D 13; E 14 and F 15. The following hexadecimal number can be converted as follows:

```
5DA9
    16
  = 5 * 16^3      + 13 * 16^2    + 10 * 16^1   + 9 * 16^0
  = 5 * 4,096     + 13 * 256     + 10 * 16     + 9 * 1
  = 20,480        + 3,328        + 160         + 9
  = 23,977
          10
```

Hexadecimal and octal numbers can be used in C exactly like decimal numbers. Hex numbers use the format specification `0x` (or `0X`) to indicate that they are base 16. For the octal system only a leading zero is required. Some examples:

```
0X5DA9
0xFFFF
0612
0x5da9
0x123
0815
0X5da9
06543
```

One of the above combinations is wrong. Examine the numbers again carefully. The error is hidden in the innocent number combination 0815. With the leading zero it should represent an octal number. There is no digit with the value 8 in the octal system.

The response depends on the compiler. The compiler can issue a message that a wrong number was entered, or accept it as a decimal number. The Lattice C compiler converts the number from octal into decimal notation. This produces something entirely different, namely 525_{10} which is equal to 1015_8

The user would soon get tired of entering every number for conversion. A good C implementation does that conversion for you.

To write a program to convert numbers from various bases into the decimal system, the procedure must differ slightly. Nothing is simpler than constructing a loop to save typing time. Starting with the last position (9 in the last example), multiply it with the value of the position. The value at the last position is then `9 * 1 = 9`. The next position has the value 16, and the variable containing this value with the base (16). The next position is therefore `10 * 16 = 160`.

All computed intermediate results are added in a separate sum. This is the same path as the manual procedure with the exception that every step again is divided into smaller steps. The user doesn't have to understand how the calculation works, since you are trying to learn C and not mathematics.

9.3.1 Conversion Program

Analyze the following listing on the basis of the explanations already provided. If it isn't clear, a few `printf` functions inserted in the program could print out the current value of one or more variables. This makes the most important variables visible.

```
/* base-con.c  9.3.1 */
void main ()
{
  long base, collect, value;
  int index, help;
  char test[100];

  printf("Please input Base of numbering system!\n");
  scanf("%ld", &base);
  printf("Input number for conversion in base %ld
          system!\n", base);
  scanf("%80s", test);
  collect = 0;
  value = 1;
  index = strlen(test) - 1; /*New Function */
  while( index >= 0)
    {
     help = test[index];
     if(help >= 'a')       /* lowercase letter */
       help = help - 'a' + 10;
     else
     if(help >= 'A')  /* uppercase letter */
       help = help - 'A' + 10;
     else   /* Probably a number */
       help = help - '0';
  collect = collect + value * help;
  index = index - 1;
  value = value * base;
  }
 printf("%s(%ld) = %ld(10)\n", test, base, collect);
}
```

The program uses the `long` data type a lot. This can also be noted in the format specification `%ld` for the input and output of these variables. The `scanf` function which reads a number as a string has something new. After the percent sign appears an 80, followed by the format specification `%s` for the string. This value between the percent sign and

the format assignment tells the function the maximum number of characters permitted. In this case the string cannot be longer than 80 characters (+ 1 end of line = 81). In reality this does not work out quite that way. As a maximum only 80 characters are processed, but the user can write several lines. Only the first 80 characters are used.

The indication of a maximum number of places is also permitted for other data types (see the example in `scanf`). A new function `strlen` will be introduced next. It delivers the characters in a string. The concluding zero byte is not included. The result is assigned with an equal sign. The only parameter required by `strlen` is the string to be investigated. One of the examples already calculated the length of the string. The `strlen` function is therefore not very large.

In the following `while` loop the string which was input is processed. To avoid the use of the expression `test[index]` for every calculation, the character at that location is copied into the variable `help`. The user should have noticed that `help` was defined as an integer variable. Yet an attempt is made to store a character at that location. Computers view characters as numbers. Every letter has a numeric code, just like a number. The `char` variables are nothing more than small integer memory areas which, depending on the compiler, accept a value between -128 to 127, or from 0 to 255. These peculiarities of calculating characters with numbers and their codes will be discussed later.

9.4 Character Codes

After the character was made available in `help`, it is tested to see whether it was an upper or lowercase letter. These can be used as auxiliary numbers in a system whose base is larger than 10. In the hexadecimal system the letters A-F are used. When it has been determined what type of character (uppercase, lowercase or number) is available, its actual value is calculated. 9 is not equal to 9 here. Confused? You remember how a character was assigned to the `char` variable:

```
character = '9';
```

The 9 is a character which represents the number nine. This character also has a special ASCII character code. The ASCII code for the number 9 is the value 57. The assignment that follows gives the same result as the example above:

```
character = 57;
```

Calculation with the variable requires the value 9 and not the stored code 57. First subtract 48 from 57 (48 is the character code for zero (0)). This is practical since all numbers are in sequential order with the following codes.

Code	Character
48	0
49	1
50	2
51	3
...	...
57	9

The letters of the alphabet also follow this order. The table starts with A (code 65), B (code 66), etc. The lowercase letters are in a separate list. The first value there is 97 for the character `a`. Let's look at a program section:

```
char test

test = 'B';
test = test - 'A' + 10;
```

What is contained in the variable `test` after the execution of this sequence? An equivalent part appears in the conversion program. The right result is 11. In the last line, test contains the letter B with the value 66. Subtracting A from this results in 1; plus 10 is 11. That is the value that the letter B represents in the hexadecimal system.

The following line converts an uppercase letter into a lowercase letter:

```
test = test - 'A' + 'a';
```

That is much more readable than:

```
test = test - 65 + 97;
```

or

```
test = test + 32;
```

The codes are nearly identical on almost all computers thanks to the ASCII standard. ASCII assigns a specific code to each character.

To obtain an overview of the ASCII codes, the program below displays every code and its character (32-127, 160-255). The codes 0-31 and 128-159 were left out because they either have special functions (e.g., 13 is equal to \n) or do not produce anything on the screen.

```
/* ASCII.c 9.4*/
main()
{
  int i;
  printf("\n\n");
  for(i = 32; i <=127; i = i + 1)
     printf("\t%-3d %c", i, i);
  for(i = 160; i <= 255; i = i + 1)
     printf("%-3d %c \t", i, i);
  printf("\n");
}
```

These are all of the characters that can be printed with the `printf` function.

9.4.1 About the Backslash

Output using a character code displays a character which cannot be accessed with the keyboard or with the backslash. The backslash must precede the code. The compiler replaces the combination of backslash and the individual digits of the code with a single character. One hitch—the number must be entered in octal notation instead of decimal notation. The following command displays the ± character (character code 177):

```
printf("\261");
```

The number 177 decimal corresponds to 261 octal. The conversion can be avoided using a format specification as shown below:

```
printf("%c", 177);
```

The character is not used directly in the string, but goes directly to the `printf` function in the form of a character code, with the `%c`.

```
printf("The result is \2611.\n");
printf("The result is %c1.\n", 177);
```

The control character `%c` permits the output of a single character by indicating the character code, even if an integer value was passed.

9.4.2 Going the Other Direction

The following program converts decimal numbers into octal numbers. It is almost the reverse of the previous conversion program which converted numbers into the decimal system.

```
/* dec-conv.c 9.4.2 */
void main()
{
  long base, test, help, rest;
  int index;
  char result[260];

  printf("Please input number Base!\n");
  scanf("%ld", &base);
  printf("Input number in decimal system!");
  scanf("%ld", &test);
  index = 0;
  for(rest = test; rest > 0; rest = rest / base)
    {
      help = rest % base; /* Remainder of Division */
      if(help > 9) /* Letter to substitute */
        result[index] = help + 'A' - 10;
      else
        result[index] = help + '0';
      index = index + 1;
    }
  printf("%ld(10) = ", test);
  index = index - 1; /* last entry is still unused */
  while(index >= 0)
    {
      printf("%c", result[index]);
      index = index - 1;
    }
  printf("(%ld)\n", base);
}
```

10.
The Preprocessor

10. The Preprocessor

The preprocessor is a part of the compiler program which first processes the source code. It accepts source code text as written. There are some special directives which force the preprocessor to make changes in the program source code text. After the preprocessor has done its work, the part of the compiler responsible for the translation uses this "processed" version of the source code text. This version can appear quite different from the listing.

To differentiate the preprocessor directives from other C statements and functions, there are two important guidelines:

1.) All directives begin with the # character

2.) All directives begin in the first column

10.1 #define

Let's first consider the most important and most often used preprocessor directive: `#define`. `#define` replaces a certain character string with another string. The preprocessor exchanges the two text strings. Let's think about what the text replacement could be used for.

Assume that a constant is used during calculations. For example during the calculation of a sales tax, a certain percentage (4% perhaps) appears regularly. If this percentage is used 10 to 20 times in a program and the sales tax percentage changes, a change in the program can become difficult. It can also lead to errors. Perhaps a wrong value would be returned if the number 4 appeared elsewhere in the program. An entry can be missed during the changes. It is simpler to use the `#define` directive. An application would appear as follows:

```
#define TAX 4
```

Up to this line the text `TAX` can be used which is then replaced by the preprocessor with the text `4`. Also the following line could be used:

```
printf("TAX-rate %d", TAX);
```

The preprocessor passes to the compiler the following substitute line:

```
printf("TAX-rate %d", 4);
```

Nothing has changed within the parentheses. This is good since it would be impossible to output a string such as TAX on the screen. Nothing inside the quotation marks can be touched by the preprocessor.

Defines are always used in a large program. The following program explains the usage of #defines. What the program produces can be seen readily in the listing:

```
/* define.c 10 */
#define BEGIN 1
#define END 100
#define STEPS 2

void main()
{
      int i;

      printf("\n");
      for(i = BEGIN; i <= END; i = i + STEPS)
        printf("%5d", i);
      for(i = END; i >= BEGIN; i = i - STEPS)
        printf("%5d", i);
        printf("\n");
}
```

Even in a small program the use of #define directives can enhance the readability of the program. An example is marking the end of a string with a null byte. This null byte is also called end of string. With the abbreviation EOS, it's an often used #define. The definition appears as follows:

```
#define EOS '\0'
```

That is more correct than simply indicating a 0. The entries of a string are considered individual characters. It is therefore good C style to use data type assignments. The single quote mark informs the compiler that a single character is used. The backslash followed by the octal value indicates the character code (see Section 9.3).

The number zero in the octal system, the decimal and other number systems is always zero. A conversion in this case isn't difficult. Using the character with the code zero, or the code directly (zero) in the assignment is of no consequence. In future programs which use strings, the definition of EOS should appear in one of the first lines.

If the reader thinks that the subject of #define is now finished, he is wrong. The many capabilities which are provided with the #define directive, will be discussed in more detail in a separate chapter.

10.2 #include

Another important preprocessor directive is `#include`. A file can be combined with the source file during compilation with this directive. This is similar to appending a file to the current file (<Esc><IF> from ED) and then saving the appended file. The compiler does not differentiate where the definitions originated, because for the compiler only one file exists. This preprocessor directive is ideally suited to include multiple `#define` directives into the program. Assume that the following `#defines` were stored in a file with the name def_new.h:

```
#define EOS '\0'
#define MAXLEN 81
#define EOF -1
```

If you have a source code text that uses these `#define` directives, you don't have to re-enter them. All you have to do is `#include` the file def_new.h:

```
#include "def_new.h"
```

The file extension of .H stands for Header file. This ensures that all `#define` directives are available throughout the listing. It is not a requirement, but should be done anyway. Although this preprocessor directive can appear at any place in a file, it is better to include it at the head of the source code.

The filename is written between quotation marks. In this case, the compiler searches in the directory where the source code is located. You can also enter the include file within greater than and less than characters:

```
#include <def_new.h>
```

stdio.h

The compiler assumes that the file is now located in a subdirectory in which all .h files can be found. The path to this subdirectory passes to the compiler during the start. There is a series of these files which are waiting to be used. One of the most popular of these files can be found under the name `stdio.h`. This stands for STanDard Input Output Header file. In Lattice C, it is in the `include` directory. This file can be examined using the ED editor.

11. Abbreviations

11. Abbreviations

We said earlier that C is an ideal language for lazy people who don't like to type. This is still true, since C lets you compress many functions into smaller packages using abbreviations. This chapter describes the art of abbreviating code in C.

C abbreviations help save typing time. Let's start with the simplest abbreviations—those used in arithmetic operations. The equation below may look fairly familiar to you. Believe it or not, this can be converted to a shortened form of the same equation:

```
number = number * 4;
```

What could be saved here? The variable `number` appears twice. This doesn't have to be so. The C language allows you to abbreviate the equation to the point where you only need to use the variable `number` once instead of twice:

```
number *= 4;
```

Every time you use the same variable during calculation and for storing the result of the equation, you can use this short form instead. The multiplier gets moved to the left side of the equal sign. The multiple of the variable `number` remains to the right of the equal sign.

The above abbreviation becomes most effective when using long variable names. In addition, it helps decrease the number of typing errors (the less you type, the fewer mistakes you make). For example, look at the following abbreviation:

```
the_user_input[index] += '0';
```

The above abbreviation corresponds to:

```
the_user_input[index] = the_user_input[index] + '0';
```

Another advantage is the speed difference between the long version of the code and the abbreviated version of the code. The execution speed of each compiled code is different; the abbreviated version executes in less time than the original. The compiler knows what values are used and where to store the result. This can save a lot of unnecessary calculation time.

Implementing operator abbreviations is fairly easy to do. All arithmetic operators can be changed into abbreviated form as seen in the following list:

```
+=
-=
*=
/=
%=
etc.
```

Consider the following expression. Can you see any possibilities for abbreviating the code?

```
value = value * (5 + number);
```

The line is already written in such a way that the operator to be abbreviated becomes immediately obvious. It is the multiplication operator. So, if you change the equation into abbreviated form, the source code looks like this:

```
value *= (5 + number);
```

Since usually the right side of the equal side is calculated first, no parentheses are required. Therefore, the final version of the short equation looks like this:

```
value *= 5 + number;
```

Now for the same thing in reverse. The operator and the named variable can be attached to the terms, using parentheses. The following equation also has potential for becoming an abbreviated version:

```
var *= number1 - number2;
```

corresponds to:

```
var = (number1 - number2) * var;
```

11.1 Increment and Decrement

Shorthand notation can be carried a step beyond arithmetic operators. The operators ++ and -- increment and decrement a variable's contents by one. The ++ operator increments the specified variable by one, and is therefore called the *increment operator*. The -- operator (called the *decrement operator*) acts in the opposite manner and decrements (decreases) the specified variable by one. These operators appear as follows:

```
main()
{
      int i;
      i = 1;
      while(i++ < 100)
        printf("%d ", i);
}
```

This short program is deceptive. Up to the `while` loop everything is clear. The `i` variable contains the value 0. Now comes the expression:

```
i++ < 100
```

First the computer sees if `i` is less than 100. Then it increments the value of `i` by 1, regardless of the results of the test. This corresponds to the following `if/else/while`:

```
if(i < 100)
  condition = 1;
else
  condition = 0;
i = i + 1;
while(condition)
 . . . .
```

Here all four listed directives are executed within the parentheses. That makes the increment operator very powerful.

It gets better. The increment and decrement operators can be placed before or after the variable, to serve different purposes. The location is important, as you'll see. A simple example will illustrate:

```
i = j++;
```

Assuming that `j` contains the value 3, `i` also contains 3. Then the value of `j` is incremented by one to 4. In contrast, the next line places the operator on the other side of the variable:

```
i = ++j;
```

With the same assumptions, the content of `j` is incremented to 4 and then the variable `i` is assigned that value. Both variables now contain 4.

Remember, if you precede a variable with an increment or decrement operator, the content of the variable changes before it is used for additional tests. If the operator follows the variable, first the current value is used and then the variable is incremented or decremented. It is important to remember this small but decisive difference. Examine the output of the two programs on the screen. The first number which appears there is two. That is clear since the starting value of `i` was one which was already incremented inside the loop head with `while`. For this reason, the `i` at the time the `printf` occurred already had the content 2.

These operators help to write fast and compact programs. They are even more efficient than the abbreviations using the equal signs.

11.2 Initialization, Definition, Declaration

Initialization

These three concepts are very important for the C programmer and should not be confused. Let's begin with *initialization*. It describes the first assignment of a value to a variable. After this point you know what the variable contains. Before the variable can be initialized, it must be defined or declared. Definition takes forms similar to the following:

```
int index;
char string[80];
```

When the compiler reaches this point, it knows the variables and sets aside the necessary memory area for them. An integer value generally requires two bytes. The variable `string` requires 80 bytes since every `char` element requires one byte. Functions can also be defined. Up to now only the definition of `main` was mentioned. If you declare a function or variable, this only tells the program that such a variable or function was defined somewhere. For this reason no memory is allocated.

Definition

Here's a tip for saving lines of code. Variables can be initialized during definition. That saves one program line:

```
int index = 0;
```

Any expression can be assigned to the newly defined variable. The string length which was determined with the `strlen` function can be used during initialization as follows:

```
int end = strlen(string) - 1;
```

Of course `string` must have been previously defined.

Some coding can seem exaggerated, but there is no limit to your imagination.

```
long middle = 4*((strlen(string1)+1)/2+1)-strlen(string2)/3;
```

Declaration

If you write a large program stored in several modules (files), a variable used by all modules only requires a single memory allocation. The definition is in one file and all the other files only contain the corresponding declaration. Declaration is made with the C word `extern`. The compiler knows that the memory was reserved externally through another file. Otherwise the linker stops linking. Example:

```
extern char pass_word[80];
extern int error_nr;
```

The example above shows that the data type must also be specified. This provides all the information necessary to the compiler about the variable. The function declaration is similar.

```
extern long atol();
```

If you define the function in the same file, the `extern` can be omitted. The declaration is still required since the compiler knows the function names and their data types only at the end of the file.

11.3 Multiple Assignments and Directive Value in C

Source code can also be abbreviated by using multiple assignments. If several variables are to be assigned the same value, individual assignments were previously required for every variable. The same value was given for each:

```
begin = 0;
sum = 0;
```

However, the following line performs the same function:

```
begin = sum = 0;
```

The assignment is from right to left. First 0 is assigned to `sum` and then `begin` gets the content of `sum` which is `0`. A term with more simultaneous assignments could be enclosed in parentheses, which would make the sequence more readable. Here's one version:

```
a = b = c = d = 2;
```

This version shows added parentheses for readability:

```
a = (b = (c = (d = 2)));
```

Individually expressed, the two above lines correspond to the expression:

```
d = 2;
c = d;
b = c;
a = b;
```

Multiple assignment is possible since every expression has a value (the result of the last operation performed). For example, the value of (d = 2) 2, of `(index = strlen(string)) strlen(string)`. Except for large initializations of variables, the value of an expression can be used almost everywhere. It also shows who knows C well. The shorter formulation will identify the professional.

Examples can show this better. Here are some more values for expressions:

```
(2)                 2
(a)                 a
(a *=3)             a*3
(a=(b=(a+2)-3))     a-1
```

The last example must be dissected into its components to reach the same result.

```
(a=(b=(a+2)-3)))
(a=(b=a-1 ))
(a=(a-1 ))
(a-1)
```

Of course the advantages of the multiple assignment can be used during the definition and initialization. The following line is permissible:

```
int start = value = 0;
```

The variable `value` must be predefined and initialized (very important) which is the case here.

12.
Functions

12. Functions

You read in the introduction that a C program sometimes consists of many different functions. Up to now only one has been defined (the `main` function). It's time to start writing programs which contain several functions developed by you.

Function structure

First, the formal structure of a function definition. You must specify the function name, preceded by the data type returned by the function. The name must correspond to the usual rules for variable names.

Parentheses containing the arguments follow the name. If no such values exist (e.g., the `main` function) none can be indicated. If such arguments are expected, these variables must be declared. The values are important since most functions get information from other functions which are then processed. Then follow the executable commands, also enclosed in parentheses.

Let's look again at a simple version of the `main` function:

```
main()
{
     ...
}
```

The first item to be encountered according to specifications is the data type which the function returns. Since the `main` function doesn't return any values to the calling program, the data type is omitted. The word `void` usually appears preceding a function that returns nothing.

Next the function name (`main`) is specified, followed by a pair of parentheses. Since no values are passed to the main program, no data appears between the parentheses. The variable declaration is also omitted, since nothing is passed. Then follow the other executable instructions within the braces, which up to now was the complete executable program.

12.1 Functions with Arguments

The next step is to dissect the program into individual tasks. You can write a short function for every partial task. For example, a function to compute the square of a value requires no great mathematical training:

```
double square(x)
float x;
{
      double q_number;
      printf("The square of %f\n is ", x);
      q_number = x * x;
      return q_number;
}
```

The `square` function

The above routine defines a function named `square` which in turn returns a `double` value to the calling program. As a parameter to be passed, a `float` value called `x` is required. At the end of the routine a new C word appears, the `return` keyword. It delivers the desired result of the specified data type to the caller and also ends the function.

It is important that no semicolon follows the function name. There must be a semicolon after each parameter declaration. This differentiates a function definition (without semicolon) from a function call (with semicolon). The following line identifies that a function named square is to be used by the main program:

```
double square();

main()
{
      float value = 3.0;
      double result;
      . . . .
      result = square(value)
      }
```

The names of the parameters passed by the calling function need not be identical to those of the called function. However the data types must be the same. Notice the line in which the `square` function is declared as a function which returns a `double` value.

The declaration can be omitted if integer values are returned. The same is true for the definition of a function. If the function returns integer values, a data type need not precede the function name. This is only possible with data type `int`. All other types must be declared and supplied with the proper data type during the definition. If one of these data types is contradictory (perhaps because the declaration forgot a `double` function) the resulting values will be wrong. While the C language permits much freedom to the programmer, but this can cause much trouble.

12.2 Functions without Return Values

Some functions return no values. These functions can be declared as `void`, if the compiler has implemented this C keyword. This can improve the speed somewhat since the parameters need not be prepared for the calling function. Even that may be omitted, which is the reason why some C compilers don't define the `void` type.

```
/* key.c 12.2 */
void key(string)  /* Without Return value: void */
char string[80];
{
  int i;
  for(i = 0; string[i] > 0; i++)
    printf("%c", string[i] + 1); /* From 'A' make 'B' */
}

void main()
{
  char text[81];
  void key();

  printf("\n\nPlease input some text!\n");
  scanf("%80s", text);
  key(text);
  printf("\nin the original it was %s\n", text);
}
```

The new defined functions are called exactly like the routines from the libraries. In this example the `main` function stands at the end of the file. The routine named `key` is declared as a function which returns nothing, or `void`. That is important since the definitions would contradict themselves during usage in `main`. If the function had not been declared, the compiler would assume that it should return `int` objects. It returns nothing.

12.3 Other Functions

Another function which does not return a result is `strcpy`. This routine copies strings, and performs general string handling. Even though it's included in every compiler's library file, it is interesting to see how it can be programmed.

12.3.1 strcpy-Version 1

This copies one string to another. Unlike the previous example, you don't know how many entries are in each string. This can be omitted. It is enough for the compiler to know that it will get a string.

In the routine itself, a counter tests all entries. They are copied until the routine reaches the EOS character (the end character must also be transmitted).

```
#define EOS '\0'

strcpy(to,from)
char to[], from[];
{
      int i = 0;
      while((to[i] = from[i]) != EOS)
      i++;
}
```

The function is indifferent to the memory requirements of the array, since it doesn't have to set aside any memory. The `strcpy` function works directly with the strings passed to it from the calling function. The strings may be of different lengths.

What do you think of the termination conditions in the `while` loop? The position of the parentheses makes the processing clear. First is the assignment of `from[i]` to `to[i]`. The expression in parentheses also has the value `from[i]`, and also the character which was copied. This is now compared with the end code character. If you copy the EOS, the condition is no longer true and the loop terminates. Otherwise it increments the current counter and remains in the loop.

The actual loop body has only a peripheral role. The main action occurs in the ending conditions. Experiment with this function. Notice that the string into which the copy is stored appears first. Here is a complete example program:

```
/* copysrt.c  12.3.1 */
#define EOS '\0'
#define MAXLEN 81

strcpy(to,from)
char to[], from[];
{
     int i = 0;
     while((to[i] = from[i]) != EOS)
     i++;
}

void main()
{
     char s1[MAXLEN], s2[MAXLEN], s3[MAXLEN];

printf("Your name, please\n");
     scanf("%40s", s1);
     strcpy(s3, s1);
     strcpy(s2, "TEXT IN s2");
printf("Therefore %s, in s2 is \"%s\".", s1 ,s2);
printf(" I hope %s, that everything is clear!\n", s3);
}
```

The `strcpy` function can be used to initialize strings since the following expression is not permitted in C.

Wrong:

```
main()
{
     char text[20] = "This_is_text!";
     . . . .
}
```

Right:

```
main()
{
     char text[20];
     strcpy(text, "This_is_text!");
     . . . .
}
```

This copies the complete string into the variable text.

12.3.2 strlen

You used the `strlen` function earlier in this book. It is simple to write and return a value. The passed length of the string is a whole number and should be an integer value.

```
strlen(string)
char string[];
{
      int i = 0;
      while(string[i])
      i++;
      return(i);
}
```

A nice short function! The expression `string[i]` is always the content of this element. This means that the expression is only 0 (false) when the end character `\0` (EOS) has been reached. The counter which corresponds to the length of the string passes to the calling function as an integer value through a `return` directive. This function doesn't have to be declared in the calling function because it returns an `int` value.

If the `return` directive passes data, it must be assured that the value has the proper data type. If the function definition states that the routine returns a `char` element, there should be a variable or constant of the `char` type. Some compilers will not tolerate such mistakes and will issue an error message. Others are indifferent and convert the result into the data type indicated in the definition. It's better to do it right in the first place.

13.
Arrays

13. Arrays

Up to now strings have been used as if they were a special data type. A string is actually multiple `char` entries. A string of similar objects is called an *array*. Arrays can also be made using `int` or `float` data types as well as `char` types. Any elementary data type can be stored in an array. Several similar variables can be accessed through a single identifier. A single element is accessed by using a subscript called the index (counter). The definition of a long array differs little from string definition:

```
long value[20];
```

This line reserves 20 elements of type `long` for the variable `value`. To indicate the end of a string, the last entry contains the value 0, i.e., assigns the escape sequence `\0`. For this reason, the definition of a string (character array) requires one element more than needed for the actual string. No such requirements exist for other array types: Only as many entries are defined as required by the data. A value assignment of one element is possible only by providing the index. For example:

```
value[0] = 4711;
value[1] = 707;
value[2] = 31415;
```

The index value of the first element always starts with 0. Using this method, you can create a string one character at a time:

```
char string[80];

string[0] = 'O';
string[1] = 'K';
string[2] = '\0';
```

This tedium can be avoided by using the `strcpy` function. The assignment sequence above would store the string value `OK` into the variable `string`, and is terminated with the usual end code `\0`.

Again the difference between a single character and a string of characters should be emphasized. The difference between "K" and 'K' is that "K" is a string, while 'K' is a character. If a letter is enclosed in "quotation marks" like a character string, it is a string. It is also terminated with a `\0` so that "K" consists of two characters, the K and `\0`. However, 'K' is only a single character. This condition must always be observed since all operating system routines assume that the string terminates with `\0`. The last element that may be accessed has an index value of 79, according to the declaration above of `string[80]` (counting starts with 0).

13.1 Multi-dimensional Arrays

Up to now we've been using one-dimensional arrays, i.e., variables which use a single subscript. Multi-dimensional arrays have elements like a one-dimensional array. However, multi-dimensional arrays have multiple elements. For example, if you were designing a chess game in C, you'd might use an 8 x 8 array for chess board data:

```
int field[8][8];
```

Both elements are of course between 0 and 7. You need two subscripts to access a single field:

```
printf("Content of Line 2 Column 4 %d\n",field[1][3];
```

You can define an array with up to five sets of elements:

```
long content[4][5][6][7][8];
```

Please observe that arrays can quickly occupy large amounts of memory. The array above would require 4 * 5 * 6 * 7 * 8 * 4 (size of a single long element) bytes (26,880 bytes or 26.25K).

Data can only be stored sequentially in memory. The user must get away from the notion that a two-dimensional array is located in two tables which are one in front of the other. How would a five-dimensional array be stored? Since all elements are stored in a long series (one-dimensional) there is a rule which must be followed. The first index changes only when all elements which belong to its group are stored. During the second index that occurs more frequently and the last index changes with every element. This concept is easier to understand in a listing which shows the position of the entries in memory. Assuming a definition of `int pos[4][3];`, entries in memory are:

```
[0][0]
[0][1]
[0][2]
[1][0]
[1][1]
[1][2]
[2][0]
. . .
. . .
[3][1]
[3][2]
```

To conclude this chapter we want to present a program which operates with arrays, and touches on many topics previously discussed. The program tests a series of numbers, passes them to a routine which adds them and receives a sum back. Then it makes statistical evaluations to determine if it's worth storing the values. An array stores the data entered. There are also some tricks which should be examined closely.

```
/* array.c  13 */
#define FALSE 0
#define TRUE 1
#define MAXENTRY 20

long total(); /* Declaration of the function */

void main()
{
  int i, number, end = FALSE;
  long sum, data[MAXENTRY];

  for(i = 0; i <MAXENTRY && !end; i++)
  {
      printf("Enter %d. value: ", i+1);
      scanf("%6ld", &data[i]); /* 6 digits limit */
      if(!data[i])
          end = TRUE;
  }

number = i - end; /* If last data 0, than one less */
sum = total(data, number);
printf("The Sum of all %d values is %ld\n", number, sum);
  printf("Deviation from Average %.9lf:\n", (double) sum
/ number);
for(i=0; data[i] > 0; i++)
  printf("Value %d: %5.9lf%%\n", i+1,
  data [i] * 100.0 / ( (double) sum / number ) - 100.0);
}

long total(array, cnt)
long array[];
int cnt;
{
  long sum = 0;

  while(cnt--) /* short and precise */
    sum += array[cnt];
  return sum;
}
```

Lattice

The library for mathematical functions and floating point numbers must be linked with the standard library. Example:

```
lc -Lm  array
```

Aztec

If you work with the Aztec compiler, the library for mathematical functions and floating point numbers must be linked with the standard library `c.lib`. Example:

```
cc +L array.c
ln array.o -lm -lc
```

First some information on the program. To make it more secure, only 20 entries are permitted. The `#define MAXENTRY` allows you to adapt the program to larger input.

The declaration of the `add` function is important. Since this function uses `long` values, the compiler must be told this. The declaration can be performed within the `main` function.

The `&&` operator ends the `for` loop which connects two tests logically with an AND. If not all entries are occupied, and the variable `end` is unequal to 0, the loop executes.

Negation

The negation operator converts `end` into the logical opposite. At the beginning the variable contains the value 0 so that the expression becomes `!end 1`. The reverse occurs when `end` is set to 1 and the negation `!end` is used to leave the loop. This happens when the user enters the number 0, indicating the end of the input.

The first entry in the array requires the index 0. Since the count usually starts at 1, a 1 is added to the current index during text output. In the formulation of the `scanf` function, the following is most important:

```
&entry[i]
```

If another array was input, no `&` character appears. That was the big exception. Since `entry[i]` and not `entry` was written, this is not an array, but a perfectly normal `long` variable. During the `scanf` routine it is equipped with the `&` like all elementary data types. The fact that this variable is in a long string of similar elements doesn't concern the `scanf` function.

The following `if` test also merits closer examination. This is a typical case of C abbreviation. The test should pass the value 1 to the `end` variable if the current input was a 0. The following shows this:

```
if(entry[i] == 0)
. . . .
```

If `entry[i]` contains a zero, this expression is also zero. With the help of the negation operator a true result is obtained. Especially for the test `== 0` or `!= 0`, the abbreviations are often placed in the location where you would expect to find an explicit value. It doesn't complicate the matter, but the user must know what is hidden there.

When the loop finishes, either because 20 entries had been made, or the last entry was 0, the total number of the stored data is calculated. The `add` function gets the necessary data (the array with the input and the number of values to be added). This routine returns the sum. With this information the deviation of each entry from the average can be calculated. If the task of the program was only to add a series of numbers, no arrays would be needed but all entries could be summed after their entry.

14.
More about Loops

14. More about Loops

This chapter takes you through a few of the fine points of using loops in C programming language. You've already seen `for` loops and `while` loops.

It also demonstrates some refinements to the `for` loop; statements which help control loop programming (`break` and `continue`); and a function for switching around within a loop (`switch`).

14.1 More About the `for` Loop

We described the `for` loop earlier in this book. Now we'll look at the limitations and flexibility of the `for` loop.

Individual components of the `for` loop are separated by semicolons. Several statements can be placed within the initialization and the increment expressions. They use commas as separators, instead of semicolons. This is how the `for` loop can be used:

```
for(sum = 0, i = 1; i <= 20; i++0
sum += i;
```

or also

```
for(i = 1, j = 0; i < 10; i += 2, j+=3)
. . . .
```

This is the usual construction for a `for` loop. Since C permits other variations, this example is presented:

```
for(printf("Now we start!"); ; printf("Bang\n"), i++)
  if( (c = input()) == 'e')
   break;
```

The text "`Now we start!`" appears at the beginning of the loop. A test is then made to determine if the condition located between the semicolons is true. This is always true since nothing is entered there.

You may recall that, under every condition, a null value is always considered a false condition. Everything else is considered logically true. The condition in the loop is always true. The only way to stop the program is to press the <E> key, provided there is an input function.

14.2 break

If the test for `if` is true, the `break` statement is carried out. The `break` statement ends the currently executing loop immediately and forces the program to continue with the statement that follows the loop that just ended. The `break` statement is the only way to break free of a loop at any time.

Look at the program in the preceding section. The increment proceeds in an unusual fashion. A `printf` call can be found there. This `printf` executes at the end of each loop execution (notice that not much remains of the original construction). An endless loop, which doesn't have an initialization, a test or incrementation, would appear as follows:

```
for(;;)
  {
   . . .
  }
```

A `for` loop can always be replaced with a `while` loop, and vice versa. The general format is:

```
for(term1; term2;term3)
  {
     other directives
  }
```

or

```
term1;
while(term2)
  {
     other directives
     term3;
  }
```

14.3 continue

The `continue` statement does the opposite of the `break` statement. Instead of leaving the loop immediately, the program jumps to the next directive in line for execution after the last directive within the loop is processed. For the three types of loops this is:

1. The body of the `while` loop (within the parentheses)
2. The incrementation of the `for` loop, therefore `for(...; ...; continue)`
3. The directive after `do`, in `do...while`

Example:

```
calculate(field)
double field[];
{
   int i;

   for(i=0; i<number; i=i+1)
      {
        if(field[i] == 0.0)
          continue;

        continue here!
      }
}
```

If an entry within field should have a zero value, the `continue` directive then executes. The program continues at location `i=i+1` as if the loop block had been terminated.

14.4 The `switch` Directive

This directive allows you to handle several similar comparisons. This is presented in the following short program:

```
/* switch.c 15.4 */
void main()
{
 int number;

 printf("Please input a number!\n");
 while(1)
  {
   scanf("%d", &number);
   switch(number)
     {
      case 9:

        printf("Larger than 8\n");
      case 8:
        printf("Larger than 7\n");
      case 7:
        printf("Larger than 6\n");
      case 6:
        printf("Larger than 5\n");
      case 5:
        printf("Larger than 4\n");
      case 4:
        printf("Larger than 3\n");
      case 3:
        printf("Larger than 2\n");
      case 2:
        printf("Larger than 1\n");
      case 1:
        printf("Larger than 0\n");
      case 0:
        printf("Number!\n");
        break;
      default:
        printf("Single number only!\n");
     }
    if(number == 4711)
     break; /*Leave endless loop */
  }
}
```

The `switch` statement is given the value to be tested (`switch(c)`). Within the block of statements, this value is compared with the values behind the keyword `case`. This value, which must be followed by a colon, is then followed by the statements to be executed. If the

comparison is positive, if all values agree, the statements following `case` are executed. If the comparisons are negative, the next comparison is tested and all statements to the next `case` are skipped.

The C keyword `default` permits execution of statements if no comparisons were successful. In comparison with the `if` test, `default` corresponds to the `else` branch of the `if` construction. If a test is positive, all of the following commands are executed. A stop doesn't occur before the next `case`. In order to stop this process, a `break` statement is required for each `case` statement.

If the character passed for the test is for example a 5, all directives (also those behind `case` 4, 3, etc.) are executed up to the next `break` directive. This causes the direct termination of a loop, or in this case the `switch` directive.

Even if several directives are executed behind a `case`, parentheses are not required. With `switch` all elementary data types except for floating point numbers can be compared.

15.
Pointers and Addresses

15. Pointers and Addresses

This chapter discusses the most important components of the C language. *Pointers* are loved by some and hated by others. In a discussion of the advantages and disadvantages of C, inevitably the word pointer will be mentioned. It is possible to write fast and short routines with pointers, but some programmers who have never worked with pointers are completely confused by them.

15.1 Addresses

Let's start slowly. The pointer concept has close connections to the address concept. During the call of the `scanf` function, the `&` (address) operator had to be placed in front of most variables. This construction allows the determination of a variable's memory address. All data, whether floating point numbers, integers or characters, are stored somewhere in the computer. The position where variable data can be stored is determined by a number (the address). In general, an address can be compared with the house number on a long street. This number is obtained from the variable which is preceded by the `&` character.

Assume that the variable `a` was defined and is stored starting at address 100. The expression `&a` would return the value 100. Why are addresses required, if you can work without them?

The user who experiments with his own functions, may soon find that the called function should pass more than one returned value to the calling function. It is also difficult to change the content of a variable defined in another function. Consider the following section from a program:

```
{
  int number = 6;
  change(number);
  . . . .
}

change(newnum)
int newnum;
{
   newnum = 5;
}
```

The `change` function receives a copy of the content of number only during the call. If this function changes the value of the variable `newnum`, the original, which is in the calling function, remains unchanged. This was already used in a program. Examine the program which calculates the sum of individual array entries. The variable `amt` decrements to zero, while the variable `number` is used later for calculation of the average.

Now we have a way to pass the address of the variable. The calling function can access them directly and the function does not contain a copy of the variables. In what data type should this address be stored?

15.2 Pointers

Principally the address could be stored in an `int` or `long` variable. This depends on the size of the `int` type and the processor, as well as how many bits are required for an address. The Amiga requires 32 bits, a long value. Not all compilers offer this capability. C may be flexible, but some compilers are better than others. Storing addresses in `long` variables is not good programming because the programs may not be portable to other computers. It is better and safer to use the data type adapted for it, the *pointer*. A pointer is marked by the special character "*". Since the data type is indicated during the definition of a pointer, the pointer is more than just a replacement for the `long` variable, What function does the pointer serve? As mentioned, it should accept an address. With this address it can access an object, here the content of a variable. During the definition the pointer obtains additional information about what data type is involved. It knows what values it points to. For example:

```
char text[80];
char *pointer; /* Define pointer to char-elements */

text[6] = 'a';
pointer = &text[6];
```

The first command defines a `char` array (string). The next line is the definition of a pointer which is called `pointer`. An asterisk precedes the pointer name, which labels it as a pointer. In addition (as in all other variable definitions) the data type is indicated. In the next line, the character `a` passes to the array element with the index 6 (7th entry). Now the pointer appears, which gets the address of element 6 with the address operator `&`. Since `pointer` now contains the address of this element, it points to the character `a`. The pointer points to another variable, `text[6]`. This is also called *referencing*, and the reversal of this process is known as *de-referencing*.

Now access can occur to the letter through the initialized pointer. The next directive could be:

```
if(*pointer == 'a')
   printf(That's it!\n");
```

If the element in the memory location should be accessed, the pointer variable must be preceded by the asterisk. The expression `*pointer` is a synonym for `text[6]` (of course, only if `*pointer` points to that position). Also the change of the content of this memory location is possible through the pointer:

```
*pointer = 'b';
```

After this directive 'b' passes to the location to which the pointer is pointing instead of 'a'. Without using the array, its content was changed.

15.2.1 The Exchange Function with Pointer

Now a routine which should change the value of the calling function. The exchange function:

```
exchange(xp,yp)
int *xp, *yp;
{
      in help = *xp;
      *xp = *yp;
      *yp = help;
}
```

This function expects two pointers to the int values passed as parameters. For the exchange the first value which points to xp is saved in the integer variable help. Then the values are exchanged. The call of the function must also be changed in comparison with the previous calls since pointers to their addresses, not int values, are expected.

```
int value1, value2;
value1 = 3;
value2 = 5;

exchange(&value1,&value2);
```

Perhaps now you can understand why, in a scanf the address operator always had to be used. With this function, data are written into the variable, which is only possible with pointers and addresses.

In arrays, especially in the frequently occurring strings, access to individual elements can only occur with the index. The address for a single entry must be obtained with &array[index]. For the first element in this list the following must be constructed:

```
&array[0]
```

In C the name of an array is nothing more than the memory address of the first element so it can be abbreviated. For &array[0] can be written array. Both return the address of the first element, not its content. An array name already acts as a pointer which points to the first element. Now it should be clear why during a call of scanf, the name of the string did not have to include the address operator &. It is already the address:

```
char string[81];
scanf("%s",string);
```

It was not an exception, only a short version of `&string[0]`.

15.2.2 strcpy-Version 2

String copying is an ideal application of pointers. Through the use of pointers, the indices which had to be used during the first formulation of `strcpy`, can be saved. The following construction with pointers illustrates an example:

```
strcpy(to,from) /* Version 2*/
char *to, *from;
{
  while((*to = *from) != '\0')
      {
        to++;
        from++;
      }
}
```

Pointer increments

In the `strcpy` routine above, the peculiarity of the pointer becomes obvious. If the pointer increments by one, the pointer points to the next element. If it increments by two, it points to the element after the next. In this `strcpy` version, a character is transmitted from `from` to `to` until the transmitted value is equal to 0. At that point the expression (`*to = *from`) has the value 0. When this expression becomes unequal to 0, the `while` loop terminates. The last transmitted character is the just tested null byte which represents the end code of a string.

15.2.3 strcpy-Version 3

The previous routine would not be a C program if it couldn't be shortened. A null test can usually be bypassed and the incrementing of the pointer can be squeezed into the termination conditions. Therefore the shorter version:

```
strcpy(to,from) /* Version 3*/
char *to, *from;
{
  while((*to++ = *from++)
      ;
}
```

This should be one of the shortest and fastest versions for copying strings which could be made faster only with a special trick. More on this later.

To write a program which transmits `float` values instead of `char` values from one array to another, only one word must be changed in the formulation above. That word is `char`. In its place the data type `float` is used and immediately `float` values, which have a completely different construction and require much more memory space per element, can be copied. How is it possible?

With the definition:

```
float *to, *from;
```

the program is informed that the pointers `from` and `to` are pointing to values of data type `float`. This data type generally requires 4 bytes per entry. If such a pointer is incremented by one, for example `after++`, it points to the following element. It is located four bytes from the original element, but the compiler knows it through the definition of the pointer. Through the incrementing of the pointer by 3, the address would change by 12 bytes. In the data type `double`, which normally uses 8 bytes, this can also be used. For each increment of the pointer, the address is changed by 8 bytes. A pointer is a very nice feature.

How does the compiler process expressions such as `string[4]`, when these groups are related to each other? Since `string` is the name of the array, which in C corresponds to the first entry (`string[0]`), the compiler converts this expression into the equivalent `*(string+4)`. First the length of 4 elements is added to the address `string`. This makes the current pointer point to the entry `string[4]`. Then access to this element is accomplished through the asterisk. The parentheses are required because the pointer "*" has higher precedence than the addition (a table of precedences can be found in the Appendices). A comparison between pointer and array can be made clear with the following examples:

```
long value, data[10]; /* Defined like this */
```

Array	the same with pointers
`value = data[3];`	`value = *(data +3);`
`data[10] = value;`	`*data = value;`
`data[7] += value;`	`*(data + 7) += value;`

As shown in the program above, other operators can be used in the construction `*pointer`. For example the `*after++` directive in `strcpy` indicates that first the value, to which `after` points (`*after`), is obtained and then the pointer should point to the next field (++). Can you imagine what the following directives would do?

```
int i, *ip = &i;
i = 100;
--*ip;
```

After the definition of the variable `i` and the `int` pointer `ip`, which is also initialized here, the variable `i` gets a value assigned. The number 100 is stored in it. Now comes the big question, what does `--*ip` do?

First the number (100), to which `ip` points (`*ip`), is obtained. Then this value is reduced by one, thus 100 becomes 99. This value is not stored in variable `i`. The same result could have been obtained with the much simpler expression `--i`.

15.3 Pointer without Storage

During the use of pointers, you should note that they represent only a pointer to a certain data type. The memory locations for the individual elements must be defined separately and the pointer pointed to them.

The initialization of the pointer prevents the system from giving wrong answers or crashing. If a crash occurs when using pointers, even during the test run of a program, first check where the pointers or the array index are pointing.

A few occasional programs seem to contradict such demands:

```
main()
{
  char *text_ptr;

  text_ptr = "All point to me!";
  printf("The text is >%s<\n",text_ptr);
}
```

Where in this program is the memory space for the string? The pointer does nothing in this direction. It is stored somewhere in the program text, just like in function calls (e.g., `printf("Hello\n");`). Also this string within the function must be stored somewhere.

Attention: If the text should be changed, for example with access through `text_ptr`, the maximum length must be observed. In the string above, this is only 28 characters, with one character representing the end of the string `\0`. If 30 characters are written into this space anyway, a system crash can be expected. It is possible that behind the string, program code was stored which was overwritten. Should the computer encounter such data which it cannot understand, it will go crazy.

The name of the array symbolizes the first element in the chain. Now the question, what is the expression `field[3][2]`, if the following definition has been issued?

```
int field[5] [5] [10];
```

Is it an element of this array? If so, which one, and if not, what is the element? Examine the expression carefully. It only contains two indices, but the definition contains three. It follows so that it cannot be an extra element. It can only be a pointer which points to the first (?) element. The first element isn't `field[0][0][0]`, but the first field to which `field [3][2]` points.

Is it clear now what wonderful changes can result from forgetting an index? One element of the field becomes a pointer to a field in which the missing index is replaced with `[0]`. Therefore `field[3]` points to `field[3][0][0]`. If something like this is possible, it can be done with pointers. Later we will see some other tricks with pointers.

16.
Storage Classes

16. Storage Classes

This chapter discusses various groups of variables. These variables have different lifespans during program execution. There are four storage classes: `auto` (or `local`), `global`, `register` and `static`. Each of these storage classes help your programs identify which C functions recognize which variables, and determine how long the functions should use these variables.

16.1 Auto

auto variables

Even though the name is unfamiliar to you, you've been using the `auto` (`local`) variables all along. The `auto` variables represent the default storage class in C language.

These variables belong to the `auto` class because they are automatically defined every time a function is called. On the function call, C allocates memory space for the `auto` variables. The lifetime of an `auto` variable is limited to the function in which the variable is declared. After the function is abandoned through return, or the last brace of this function is reached, the memory space allocated is released and can be used for other assignments. These `auto` variables can only be used in the function for which they were defined. The content of the variable is lost and the name is not known to the rest of program.

16.2 Static

static variables

Unlike `auto` variables, `static` variables are retained until the end of the program and are not deleted after leaving the function. They do not have to be created again during a new call of the function. Leaving, which means the termination of the executing function, should not be confused with another function call within this function. Control may briefly pass to another routine, but the calling routine remains active (it's waiting for a result).

Here's an application of a `static` variable. The C word `static` appears in front of a definition. For example:

```
function()
{
   static int counter = 1;
      . . . .
}
```

During the first call of the function, the variable is defined and initialized with a starting value as in the example above. If the function is left temporarily, a new variable isn't created during the new function call because the variable still exists. Even its content remains and it does not have to be initialized again. For example, a counter in this routine could track how many times it had been called.

16.3 External

extern variable

The next storage class is the `extern` or `global` variable. These variables are defined outside the function and can be used by all functions. A section of a program would appear as follows:

```
#define EOS '\0'

int error, dummy;

main()
{ . . .
}
```

The variables which were defined can also be used by functions which are not in the source file. The linker is given a number of files for linking. These files contain functions which have already been compiled. They may need `global` variables which must be assigned the right values in their program. Such variables must be declared before using them with the `extern` function, but they don't have to be defined:

```
extern int error;
```

This permits the use of the variable in a file in which `error` was not defined.

Combinations, such as `global static` variables, are also permitted. Through this definition all functions can access the `global` variable within the source file, but the situation just described of accessing this variable with a function from another file is prohibited. The variable is only known to the source file. Functions which first come in contact with the program through the linker, have no access to this variable.

16.4 Register

register variable

The last storage class is `register`. Those of you who have some programming experience with assemblers know what this means. A processor, the most important part of a computer, has various internal memory locations. One such memory segment, which should not be confused with the RAM of the computer, is called a *register*. The number of registers which can be used depends on the type of processor used. A 6502/10 used in the C64, or in the Atari 600/800/130, has only three registers (2 registers and an accumulator). The MC68000 used in the Amiga, Atari ST and Macintosh has 17 registers. Each 68000 register is four times the size of a register in the 6502. For this reason there are almost no compilers for 6502 computers which offer the capability of register storage for variables. Of the 17 registers in the computer, only three to five (depending on the compiler) are made available for storage. The remaining registers are required for internal use.

A variable defined as `register` must fit inside a register. A 68000 register is equal to 32 bits (four bytes), which only permits integer numbers. Even if the `float` value could occupy only 4 bytes, it cannot be stored in a register. Valid data types are:

```
int             char
short           unsigned
long
combinations of the above
pointers
```

Pointers are possible since they only represent the address of an object. In the Amiga they only occupy four bytes.

There are other restrictions. The defined variable can only be an `auto` variable since it occupies a register of the central processor. They are rarely used and can be occupied only for a short period of time. After leaving the function in which it was defined, the register is released again for other purposes.

The `register` variable has a speed advantage over other variables. The program can only fully utilize this speed when these variables are used during many loop repetitions or calculations. The variable does not have to be loaded from memory into a register for every use, since it is already present.

16.4.1 Fast `strcpy` Routine

Before we present the first example, we must discuss another limitation. It isn't possible to obtain the address of a `register` variable with an & operator, because a register doesn't have an address. It is not located in RAM.

```
strcpy(to, from) /* last version */
register char *to, *from;
{
      while(*to++ = *from++)
      ;
}
```

This definition of the `char` pointer as `register` should receive the maximum speed that can be obtained in C. This could be faster only if it was written in machine language.

To test the speed advantage obtainable through registers, compile the following program. To measure the speed of the program you must use the registers as often as possible and should not use other functions since they only extend the time required. For this reason the program does nothing more than count a variable down from 5,000,000 to 0.

```
/* countdown.c 16.4.1 */
#include <stdio.h>

void main()
{
  printf("Time comparison with and without register\n");
  printf("RETURN for Start\n");
  getchar();
  printf("%cStart without", 7);
  without_register();
  printf("%cStop!\nregister routine\n",7);
  printf("RETURN for Start\n");
  getchar();
  printf("%cStart with", 7);
 with_register();
  printf("%cStop!\n\n", 7);
```

```
}

with_register()
{
  register long i =5000000;/*Count from 5,000,000 to 0 */
  while(i--)
  ;
}

without_register()
{
  long i = 5000000;
  while(i--)
  ;
}
```

The preprocessor command `#include` includes the `stdio.h` file in this C program. This file is required because the `getchar` routine is used to obtain a character from the keyboard. In the Lattice C compiler, the function is unusable because it waits for the <Return> key after every character. However, it is sufficient for the program above, so it can be used to wait for the <Return> key.

Timed by hand, the author counted 51.6 seconds without `register` variables and 24.1 seconds with `register` variables. That's impressive because it's twice as fast when the word `register` is used. It should be noted that the multitasking capability of the Amiga, which could have been performing a task in the background, was not used. This would have provided a different result.

16.5 Local

local variables

Local variables are the reverse of global variables. Different variable groups such as register, auto or static local can be defined. They are only valid in the block or the function in which they were created. A local variable has precedence over a global variable, which means that, if two variables were defined with the same name, the local variable is used. The local variable gets preference while the global variable disappears for the moment. An example:

```
/* local.c 16.5 */
int i = 1;

void main()
{
   int i = 2;
   printf("%3d", i);
    {
      printf("%3d", i);
       {
       int i = 3;
       printf("%3d", i);
       }
      printf("%3d", i);
    }
   printf("%3d",i);
   test();
   printf("\n\n");
}

test()
{
   printf("%3d", i);
     {
      int i = 4;
      printf("%3d", i);
     }
}
```

The numbers 2, 2, 3, 2, 2, 1 and 4 are displayed sequentially. In the main function a new local variable is declared so that the global variable i is no longer addressable. The following block keeps this configuration and another 2 appears. Then another block follows in which another i variable is defined. Because of this, the previous block becomes invisible to the program and the current one prevails. The result of the output is 3. After the program leaves all blocks, the hidden variables appear again. The previous variable with the value 3 is erased by leaving this block then disappears. The test function is now called and proceeds to output i. Since no local variable is known at this

point, the output of the `global` variable, which is 1, is used. Finally a `local` integer variable is activated which overshadows the `global` variable again. This ensures that the last variable defined in a block is used, and that often used names (e.g., `i` or `j`) are recognized as runtime variables in many loops as different variables.

17. User-defined Libraries

17. User-defined Libraries

One advantage of programming in C is the modular construction of programs, which can accept existing functions used in other programs. The `#include` directive lets you add external files that have frequently used functions to the current program before compiling. The compiler processes one large file instead of several small files.

Every C programmer writes his own functions at some time or another. You've already entered two functions (`strlen` and `strcpy`); let's use these. Most compilers contain these functions. However, viewing them can give us an understanding of how user written functions work.

Save these functions to your own file under the name `string.c`. You can include these functions in your own programs using the `#include` directive. The following line searches the main directory for the `include` file `string.c`:

```
#include "string.c"
```

The following line also adds the `string.c` file to the main file:

```
#include <string.c>
```

Of the two syntaxes, the second line is much more flexible than the first, since it searches many different directories for the same file.

Most C implementations have `include` files as standard equipment. Files with .h extensions contain mostly `#define` directives. You can find these functions in a library such as `amiga.lib` or `lc.lib`. `include` files can be included on demand. The syntax reads:

```
#include <file.h>
```

Before starting with `include`, first you need something that can be included. A useful function can be written to compare strings. Since strings are not elementary data types, they can't be compared with:

```
if(string1 == string2) /* This is wrong! */
```

If you wrote the variables `string1` and `string2` as character arrays, the name would correspond to the address of the first element (`&string1[0]`). Therefore, the addresses of the two arrays always differ. Since both arrays have been assigned by the compiler to separate memory locations for their char entries, the comparison is completely useless. The only case (theoretically) in which this `if` test can be fulfilled is if one or the other variable was defined as a pointer and if the pointer pointed to the same string. This method doesn't work.

17.1 The strcmp Function

You now have to write a program to compare each element of the first string one at a time with each element of the second string:

```
strcmp(s,t)
register char *s, *t;
{
   while(*s == *t)
   {
   if(!*s)
   return(0); /* End reached (*s == 0) */
   s++;
   t++;
   }
   return(*s - *t);
}
```

The strcmp function compares the characters of the s string with those of the t string. As long as the characters are alike (*s == *t), the while loop executes. A test determines whether the last character matches the \0 marker EOS (end of string). If so, both strings must be identical, since s and t end with EOS. Otherwise the pointers move to the next element and the process repeats.

If a character appears within s which differs from the t character, the while loop terminates and the difference between the two characters (*s - *t) returns to the calling program. Negative values indicate that the s string was smaller than t. Positive values mean the opposite. A null returned after the if test indicates that both strings are completely identical.

Store this function and the other two files below as stringfunc.c. Since the older strlen routine could be improved, we will use pointers this time. Instead of the indices, the pointer moves over all entries of the string up to the EOS character. The start value must be stored first so that the number of increments can be computed. That is faster than counting with an additional variable. The stringfunc.c file therefore appears as follows:

```
/* stringfunc.c 17.1 */

strcpy(to, from)
register char *to, *from;
{
   while(*to++ = *from++)
   ;
}
```

```
strlen(s)      /* Conversion to pointer! */
register char *s;
{
   register char *help = s; /* Store initial position */
   while(*s)
   s++;
   return(s - help);
   /* Difference between pointers is Element number */
}


strcmp(s,t)
register char *s, *t;
{
   while(*s == *t)
   {
   if(!*s)
   return(0); /* End reached (*s == 0) */
   s++;
   t++;
   }
   return(*s - *t); /* Difference between two strings */
}
```

Now let's see if they function properly. For this you'll use two strings which are initialized in the program.

```
/* stringtest.c 17.2 */

#include "stringfunc.c"

/* Global arrays can be initialized! */

char string1[] = "Hello!";
char string2[7] = { 'H', 'e', 'l', 'l', 'o', '!', 0};

main()
{
      printf("\nComparison of >%s< and >%s< is %d\n",
        string1, string2, strcmp(string1, string2));
      printf("Now >%S< and >%S< Result in %d\n\n",
        string1, "Huhu!", strcmp(string1, "Huhu!"));
}
```

First we will look at the expected results of the function call `strcmp`. The first call returns zero since both strings are really equal. The second function call returns -16. This number is the result of the comparison of the `e` and `u` characters. This means that the first different character in the string ("`Hello!`") is smaller than the first different character in the second string ("`Huhu`").

Array initialization is new to this program. Until now each element was stored individually. `automatic` variables wouldn't allow storage in any other form. With `global` variables, a string can be initialized

directly, or, as in the second example, every character is initialized separately. In the first example there wasn't even an indication of how many elements `string1[]` should have. This is another indication that the C language was invented for those who consider laziness a virtue. The compiler must determine the number of the character on its own. It initializes `string1` with 7 elements (don't forget the null byte at the end). Those who prefer, can indicate the value as in the second example.

If you assign the elements individually (example 2) to the fields, they must be contained in braces and separated by commas. For multiple dimensions, multiple braces must be used.

```
int field[4][4]=
{
 { 1, 2, 3, 4 },
 { 6, 3, 4, 9 },
 { 3, 4, 5, 6 },
 {12, 9, 0, 2 },
};
```

This formulation assigns `field[4][4]` the proper values where the first values { 1, 2, 3, 4 } are stored in the fields `field[0][0]` to `field[0][3]`. The inner braces are not required on some compilers and the directive could appear as follows:

```
int field[4][4]= { 1, 2, 3, 4, 6, 3, 4, 9, 3, 4, 5, 6,
12, 9, 0, 2 };
```

When some elements are not initialized, they don't have to be listed. All elements left out are automatically assigned a null.

```
in field[3][3]=     {
                     { 3, 2 },
                     { 4 },
                     { 3, 4, 5 },
                    };
```

The fields `field[0][2]`, `field[1][1]` and `field[1][2]` contain nulls. A semicolon must follow the definition. After the inner braces and the last brace there must be commas. Remember that initialization only affects `global` or `static` variables, not `auto` variables.

17.2 Itoa

Another routine seen frequently in connection with strings is `itoa` (Integer TO ASCII). It converts an integer value into the corresponding character string. When you pass the number 123, `itoa` returns the string "123" in a character array. This is very important when preparing text that contains numbers. All conversions usually performed by `printf` can also be done with user routines.

The `itoa` function requires, as parameters, an integer value which it can convert, and a string to store the result. The head of the function definition reads as follows:

```
itoa(n, s)
char s[];
int n;
```

Conversion

The modulo operator `%` performs the conversion. By dividing the number by 10 you obtain the last place. Then the code for the number '0' is added to get the first character. The number is then divided by 10 to shift it left one space and the last number drops off. The same procedure is performed on the new last position. The program section for this process appears as follows, if the index for the character array is called `i`:

```
do
   s[i++] = n % 10 + '0';
while((n /= 10) > 0);
```

The last place is converted and stored in `s` until the number which was stored in `n` has reached 0 through constant division. The sign should not be forgotten since it could cause problems for the loop (number larger than 0). The simplest process makes the number positive before the conversion and, if necessary, sets a flag for a negative value. After the completed conversion, the string returns the minus sign.

The completed processing converts the number 123 into the string "321", but only the last place is processed and stored in the string. The solution to this problem is very simple. Write another function that reverses the string. Assuming that such a function already exists (see the next section for the function) the routine appears as follows:

```
/****************************************/
/* Name:      itoa                      */
/* Parameter: n(int), s(string)         */
/* function:  Convert Integer to string */
/* comment:   Requires reverse()        */
/****************************************/

#define EOS '0'
#define FALSE 0
#define TRUE 1

itoa(n, s)
register int n;
register char *s;
{
     register int i = 0;
     register int sign = FALSE;

     if(n < 0)
       {
            sign = TRUE;
            n = -n;
       }
     do
       s[i++] = n % 10 + '0';
     while(n /=10);
     if(sign)
       s[i++] = '-';
     s[i] = EOS;
     reverse(s);
}
```

Program header

The large header contains important information. The function developed by the user should be ready for use when it is finished. After some time the function name and the parameters to be passed may have been forgotten. At that time you could consult the header with its comments. The function can be compiled independent of other functions. If the compiler permits it, it can be stored in a library. Of course the source files can be included into the current file with:

```
#include "itoa.c"
```

This increases compiler time, of course.

Since this function should be accepted in the library, it should be the latest state-of-the-art. This can be done with the `itoa` function by defining all variables as `register` variables. The define required for this function should not be omitted, even if it appears somewhat cumbersome to determine a define for a single application. It improves readability since larger programs usually access these macros.

17.3 Reverse

The `reverse` function

Now to the `reverse` function which can reverse a string passed to it. Construction of the routine doesn't present a problem. Two pointers, or indices, are needed for the beginning and end of the string. These pointers exchange their elements between themselves and are then moved toward each other. The pointer at the beginning is incremented and the one at the end is decremented. Exchange continues until the two pointers are equal, i.e., point to the same element. The routine is presented complete with a commented header.

```
/*************************************/
/* Name:reverse                      */
/* Parameter: s(string)              */
/* function: Reverse string          */
/* comment: Requires strlen()        */
/*************************************/

reverse(s)
register char *s;
{
     register int c, i, j;

for(i=0, j = strlen(s) - 1; i < j; i++, j--)
  {
     c    = s[i];
     s[i] = s[j];
     s[j] = c;
  }
}
```

The `strlen` function

The `strlen` function initializes the index `j`, which should point to the last element of `s`. Every C compiler package has `strlen` included in one of its libraries, or you can use the `strlen` function defined in the previous chapter. Both routines (`itoa` and `reverse`) should be stored in the library labeled `itoa.c` since it will be accessed later. Please note that the `itoa` routine also comes as standard equipment with most C compilers. These standard everyday functions have already been written by others.

18.
C Features

18. C Features

We have mentioned many times that C language is much more flexible than many other languages. C has many features that BASIC doesn't, and is still much simpler to use than assembly language.

This chapter examines some components of C which aren't possible in other languages. These components are partly responsible for allowing the user to take full advantage of C's speed and flexibility.

18.1 The ?: Operator

The `?:` conditional operator evaluates the first statement and returns, if the expression was true, the statement which follows. If the expression was false (=), the operator returns the second statement following the colon. This operator uses the syntax:

```
result = (expression1) ? (expression2) : (expression3);
```

If `expression1` is unequal to zero, `expression2` becomes the result. Otherwise `expression3` is returned. A concrete example:

```
c = (a>b) ? a : b;
```

This would be similar to the `if` construction:

```
if(a>b)
  c = a;
else
  c = b;
```

This term delivers the maximum of `a` and `b`. Since this is simple to formulate, this operation usually determines minimum and maximum quantity. The define:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)<(b))?(a):(b))
```

Take a look at `stdio.h` which you'll find with your compiler. There you'll find the definition.

18.2 The sizeof Function

The sizeof function

The sizeof (size of) function returns the sizes of objects (variables) regardless of type. The unit returned by this operator is defined on the basis of the char elements. The following example followed by sizeof (character) returns 1:

```
char character = 'a';
```

The result is always the number of occupied bytes for the object under investigation. The following short program determines how much memory is used by the various data types in your compiler. This will tell you if an int variable occupies 2 bytes (most C compilers) or 4 bytes (Lattice).

```
/* sizeof.c 18.2 */
main() /* Indicates memory required for data types */
{
     printf("\ndatatype\tmemory in bytes\n");
     printf("char\t\t%d\n",       sizeof(char));
     printf("short\t\t%d\n",      sizeof(short));
     printf("int\t\t%d\n",        sizeof(int));
     printf("long\t\t%d\n",       sizeof(long));
     printf("float\t\t%d\n",      sizeof(float));
     printf("double\t\t%d\n",     sizeof(double));
     printf("pointer\t\t%d\n",    sizeof(char *));
}
```

18.3 Bit Manipulation

This section describes the remaining operators, which deal with controlling individual bits.

Operators for bit manipulations exist in addition to the logical combinations. A bit (binary digit) represents a position in a binary number and can therefore only assume one of two values (0 and 1). The conversion into the binary system is similar to the conversion into the octal or hexadecimal system. The bit is also the smallest unit which the computer can use. It acts as the basis for all other numbers which can be used in the computer. For example, a byte consists of 8 bits, a word of 16 bits and a long word of 32 bits. A character can be stored in a `char` variable. Characters are stored in bytes (8 bits); a byte can accept 256 different kinds of numbers. Depending on the compiler, an integer value contains 16 or 32 bits and a long value 32 bits. Individual bits of these data types can also be accessed. These operators cannot be used with `float` or `double` variables.

18.3.1 AND

The AND operator consists of the & character. Maybe you thought this is the `address` operator. This character can be used for both purposes, but it's hard to explain why this is so. You must know the context in which it is used. If it is placed alone in front of a variable, it represents the `address` operator. If it is placed between two values in a normal arithmetic equation, then it is the binary AND.

The concepts logical and binary help you distinguish between two completely different operators. The logical AND is different from `&&`.

With AND (&) individual bits can be reset. A set bit has the value 1, a reset bit the value 0. The following table shows the connection between various bit combinations.

Combining bits using logical operators

AND			OR			EXOR		
&	0	1	|	0	1	^	0	1
0	0	0	0	0	1	0	0	1
1	0	1	1	1	1	1	1	0

According to the AND a bit is set (1) when both bits are set, otherwise the result is a 0. This is comparable with:

```
if(bit1 == 1 && bit2 ==1) /* Here is the logical AND! */
  result_bit = 1;
else
  result_bit = 0;
```

18.3.2 OR

With the (|) OR operator bits can be set. A glance at the table above will help you understand the various combinations of bits. In OR the resulting bit is set if one or both bits were set. Only if both bits are 0, is the result of OR also 0. The | sets individual bits and the & resets the bits.

A mask acts as a storage area for the bits. A mask is represented by a number placed over the value to be processed. If a variable is ORed with this mask, all bits set in the mask are now also set in the variable.

Example: Bit number 2 should be set in the variable `flags` (the count starts at 0):

```
#define MASK 4
int flags = 73;

flags | = MASK;
```

After this operation, the OR with the value 2^2 (number of the bit to be set) = 4 (sets the second bit in the variable `flags`).

A targeted resetting of certain bits sets the corresponding bits of the mask to 0. With the AND operation the desired zero bits are obtained. Example: bits 1 and 4 should be reset.

```
int flags = 37;
flags &= 0355; /*All bits except 1 and 4 are set (0-7) */
```

Every bit has its own value according to priority. For example bit 3 has a value of 8 (2^3). The table below shows the individual bit values:

```
bit Number   0    1    2    3    4   5    6    7
Value        1    2    4    8   16  32   64  128
```

Some examples for bit operations:

```
1 & 2 = 0
2 & 6 = 2
7 & 8 = 0
9 & 12 = 8
```

The last example should be examined closer in the binary system.

```
9(dec) = 1001 (binary), 12(dec) = 1100(binary)

1001
  &  1100
  ---------
     1000

     1000(binary) = 8 (decimal)
```

The same operation for OR:

```
1 | 2 = 3
2 | 6 = 6
7 | 8 = 15
9 | 12 = 13
```

The last line expressed in binary:

```
1001
1100

----
1101

1101 (binary) = 13 (decimal)
```

It is important that the characters & and && are kept separate from each other. & connects expressions bit by bit. && only makes a logical comparison from which either a 1 (true) or 0 (false) is returned. Therefore 2 & 1 = 0, but 2 && 1 = 1.

There is also a distinction between the operators | and | |. Loops and conditions may result in strange behavior if you confuse these operators.

18.3.3 Bitwise Shift Operators

>> <<

Operators for bit shifting are >> and <<. They permit bit shifting to the left or right within a field. A shift to the left (<<) by one position is the same as multiplying by 2, only it is much faster. All of this is dependent on how data and numbers are stored and processed in the computer. A shift to the right equals a division by 2. Depending on the data type, either zero bits or set bits move into the free locations. For unsigned values, zero bits are shifted in every case. For normal signed `int` values it depends on the compiler used. For positive numbers, zero bits should be added to the left, and for negative numbers, one bit. This is compiler dependent and there is no guarantee of how this works.

The number of shifts is indicated behind the operator.

```
5 << 3 = 40
```

101 (binary) shifted left by 3 bits (zero bits are shifted in): 101000 (binary) = 40 (decimal).

Use the program in the previous section for converting decimal numbers to binary numbers.

18.3.4 EXCLUSIVE OR

The EXCLUSIVE OR operator ^ is related, as the name indicates, to the OR operator. The only difference lies in the fact that both bits are set. The OR operation results in a set bit, but the EXCLUSIVE OR resets a bit. The table for the EXCLUSIVE OR is as follows:

EXOR	0	1
0	0	1
1	1	0

for example 2 ^ 1 = 3

Please do not confuse ^ with the up arrow for exponentiation. This does not exist in the C language.

Remember: The bit is set only if both bits are different.

18.3.5 One's Complement

The one's complement operator ~ requires only one parameter. All bits of the parameter are reversed. Set bits are unset and vice versa. It is recommended to use this operator only for variables which were defined as unsigned, or the sign is also affected.

```
unsigned number = ~3;
```

In the variable all bits except for the first one are set (priority 0 and 1 = 1 + 2 = 3) so that the variable now receives the following bit sequence (starting from a 16-bit integer):

```
1111 1111 1111 1100
= 65532
```

18.4 Goto

Perhaps `goto` sounds familiar from BASIC, but the C implementation is a true curiosity. This statement has a bad reputation in C, since it can destroy a well structured program. Jumping in a function can defeat the purpose of clarity in programming. Nevertheless the `goto` statement is not totally useless. It can be used effectively in error trapping. If an error occurs within several loops, which makes progress impossible, only the `goto` statement offers escape. The usual break directive can only stop one loop, not several at once. Some tests and other break commands could also terminate all loops. It is preferable to use `goto`.

The use of the `goto` statement of course assumes a label (a marked line to which the `goto` jump should be made).

```
label: printf("This is where goto will Jump!\n");
       . . . . . .
       if(error)
         goto label;
```

The labels can be defined in the program text anywhere, but must include a colon. They are only required for the `goto` statement and are formed exactly like variable names.

Note: The `label` and `jump` commands must be used in the same function. It is not possible to jump across all functions.

19. Complex Data Types

19. Complex Data Types

Now that we've listed all the important commands, we now come to the extra capabilities of C. Among these are data types which can be configured according to the needs and demands of the user.

19.1 Struct

```
struct {
char firstname[20];
char surname[30];
int age;
double income;
int sex;
 } person;
```

The `person` variable

This function defines the variable `person`. The variable `person` consists of several partial variables which are described in more detail within the braces. The first name has 20 characters, and the last name 30 characters. There are also fields for age, income and sex. Similar to arrays, many entries are collected under one name. The difference is that different variable types appear within the structure. Accessing individual parts of this variable requires more specification than arrays, which use an index. The structure uses either the . (period) operator or -> operator. An assignment of 30 to the element `age` appears as follows:

```
person.age = 30;
```

All other fields can be accessed in the same manner:

```
person.sex = 0;
person.income = 300000.0;
strcpy(person.firstname, "Rena");
strcpy(person.surname, "Bebewicz");
```

In order to use a pointer on such a construction, you must indicate the data types. All you've done so far is create a complete variable. You need a name such as `int` or `float`, through which other variables such as a pointer can be defined. If several variables or pointers are used, it would be better to create another data type which also has its own name. This can be done by indicating the type name after the `struct` statement. If it is called `person`, the uppercase letter indicates that it isn't a variable. Since variables, statements and functions must appear

in lowercase letters, and `defines` in uppercase and lowercase letters, structures can use a combination of the two:

```
struct Person {
                char firstname[20];
                char surname[30];
                int age;
                double income;
                int sex;
              } person;
```

A pointer to this structure can now be initialized for access:

```
struct Person *pointer;
```

To access one element of the structure, the expression would be:

```
(*pointer).age = 30;
```

Pointer operators

The parentheses above control the higher precedence of the (.) operator. Usually a special operator is used as illustrated below:

```
pointer->age = 30;
```

This operator, made of a minus sign and a greater than character, is easier to read. In addition, the arrow better illustrates its purpose (it "points to" something).

All operations which affect basic data types, such as defining arrays (vectors), can be used on the newly created structure. The line below provides 100 structures for storing partial variables:

```
struct Person occupant[100], *occ_upa;
```

Addressing individual entries can be done through an index (`occupant [3].income = 25000.0`), or after initializing the pointer with:

```
occ_upa = occupant;
```

Comment:

The name represents the address of the first element. The following could be written as an alternative:

```
occ_upa = &occupant[0];
```

The pointer can also be used to access the entries:

```
(pointer->income = 25000.0)
```

The pointer can be used to search the entire array. The following example sets the pointer to the first free element, provided that in an unused entry the value zero was stored in `age`:

```
while(pointer->age)
  pointer++; /* searches all entries */
```

Further applications of the `struct` directive will be discussed later.

19.2 Bit Fields

The last remaining data type is the *bit field*. Bit fields are really a form of structure definition. Unlike regular definitions, bit fields are usually taken apart rather than created. A variable is defined which consists of a certain number of bits. This variable always represents whole numbers. The value range depends on the number of bits used. This number can be calculated with the formula $2^{number_of_bits}$. These fields are arranged in `int` objects so that the maximum field width is 16 bits. This also applies to Lattice C which usually has a different concept of `int`. If a field doesn't fit into the partially occupied integer value, it goes into the next one. A lot of memory can be saved by clever selection of the field width.

```
struct {
         unsigned sex : 1;
         unsigned married : 1;
         unsigned children : 4;
       }  data;
```

As in other structure definitions, the data types are placed inside braces. To ensure that the bit field contains unsigned whole numbers, `unsigned` is used (an abbreviation for `unsigned int`). A colon separates the fieldname from the field width in bits. The definition above occupies 2 bytes (size of a 16-bit integer), but is not completely utilized. Since only 6 bits are used (1 + 1 + 4), an additional 10 bits can be assigned for other applications without requiring additional memory:

```
struct {
         unsigned sex : 1;
         unsigned married ; 1;
         unsigned nr_children ; 4;
         unsigned age ; 7;
         unsigned nr_cars ; 3;
       } data;
```

Additional data has been stored which occupies only 2 bytes. Access to each bit field occurs with the (.) operator.

```
data.nr_children = 2;
```

The limited values must be respected, since in this definition no family can have more than 15 (2^4-1) children or operate more than 7 (2^3-1) vehicles.

19.3 Unions

A special variable exists in C which accepts all conceivable data types. This `union` is dimensioned by the compiler in a way that allows it to accept all data types indicated in the definition.

```
union Universa {
                 in i;
                 double d;
                 struct Person;
                 char c[100];
               } result;
```

All data types indicated can be stored in `result`. It is useful to remember what type is stored, for example:

```
result = 2.8;
```

or

```
strcpy(&result, c);
```

The memory requirements of such a variable depends of course on the length of the largest entry. In the example above this would be 100 bytes used by array `c`. Please note that only one type can be stored in this variable. The author hasn't yet found a reason for using `union` structures instead of solving the problem with other C data types.

19.4 Enum

The C word enum defines a data type which assigns constant values to the variables. Short for ENUMeration, enum lets you assign a consistent integer number (constant) to a variable. This is useful for assigning numbers to strings. The following example assigns numbers to the first three words in color and a specific value to the black variable. Finally, the assignment to the white variable continues where the number assigned to black left off:

```
/* Definition of such a Data Type */
enum color (red, green, blue, black = 9, white)

/* variable Definition */

enum color var, *color_ptr = &var;

var = blue
if(*color_ptr == green)
   *color_ptr = black;
```

The enum type assigns an integer value to each name starting at zero and adding one for each element. With direct assignment values can be skipped. The values defined in color are:

```
red          0
green        1
blue         2
black        9
white        10
```

19.5 Typedef

The `typedef` feature can be used to create new data type names. A name assigned with this command can be used as another data type during the definition. The program below gives the word `FLOAT` the same meaning as the `double` data type:

```
typedef double FLOAT;
```

The advantage of this directive is that `typedef` permits changes in the entire program similar to `define`. Also large data types can be abbreviated with this command. Look at the example below:

```
typedef char * STRING;
```

All pointers to `char` elements can be defined simply with the word `STRING`. The `typedef` command has an advantage over the `#define` directive in that the definition of the replacement occurs in a different way. One text can be replaced with another text. A blank marks the spot in which the text replacement appears. The redefinition of type `STRING` into another word isn't possible since everything after the space behind `char` already counts as replacement text. With the `typedef` command this is just the reverse. The last string `STRING` is the replacement for the data type `char *`. It or the blanks in the middle always belong to the definition of the data type.

This concludes the introduction to C keywords. These statements can now be used in smaller programs. Practice makes perfect. Experiment by changing one or more parameters in the programs to determine the effect upon the program. Write your own new programs once you've gotten used to entering the ones already listed above.

20.
Important Concepts

20. Important Concepts

20.1 Declarations

Declarations declare the data type of a variable or function in a program. Various declarations have already been used. Directives can appear in which you have to guess the data type that was declared, unless you examine the expression for certain rules. First, some simple examples (the first three are also definitions):

```
int i;              Integer variable
float array[10]     Float variable
double *ptr;        Pointer to double element
long func();        Function which returns a long value
```

Every declaration basically contains an elementary data type (`char`, `int`, `float`), which in certain cases can be augmented by a special storage class (`auto`, `extern`, `register`, `static`), or through attributes such as `long`, `short` and `unsigned`.

```
extern double sin();  function from another file
                      returns double-value
static short digit    small Integer variable
register long i;      long-variable stored in a
                      register
```

Each name can also be equipped with combinations of *, [] or even (). The asterisk to the left of the name represents the pointer character. Parentheses and brackets appear to the right of a name. They should not contain values during declaration, because that turns the declaration into a definition ([] for arrays and () for functions). Combining all parts of a declaration can result in some complicated combinations:

```
long *field();          function which returns pointer to
                        long value
int *i_ptr[];           Field of integer pointers
float (*berech)();      Pointer to function which
                        returns float value
char *(*text)();        Pointer to function which
                        returns pointer to char
int *(*text_arr[])();   Array of pointers to functions
                        which return pointers to int
```

Complicated expressions are formed with unified rules. Going through them step by step makes the routine work later to decode such combinations. You'll need the table of operator precedence found in the Appendices. From this you can see that the parentheses have higher priority than the pointer.

Look at the first example listed above:

```
long *field();
```

Decoding expressions

First, the variable `field` is a function. Look at the left side of the line for the asterisk which defines the expression as a function. This function returns a pointer. No additional information exists beyond the parentheses. The left side of the line identifies the data type `long`. Together this information creates a function which returns a pointer to the type `long`.

After processing one side of the line, the information to the right of it must be processed (if priorities permit). The last and most complex example looks like this:

```
int *(*text_arr[])();
```

It looks complicated. It can be easily decoded by following the procedures as listed above, but the description will take a little longer.

Start with the name `text_arr`. First test which operators are processed first according to priority (to the right or left of the name). These are the brackets which indicate an array. The operator has been processed on the right; now go to the left. A pointer there indicates that this is a pointer array. The right side of the line informs you that the pointers should point to functions (the parentheses are required because of the high priority of parentheses over the asterisk). Changing sides again, you note that the function returns pointers. Since on the right side there is no additional information, continue on the left with the data type. There it shows that the pointers point to integers. This line declares an array of pointers to functions which returns pointers to `int`. Complicated expression, complicated sentence; but simple to analyze.

Only data types which cannot be used in definitions as values for passing are prohibited. For example, no function can be declared which should pass structures, arrays or functions. Pointers to such objects are permitted and are the only way to access this information.

Comment: Some new compilers also permit structure passing. This may differ from one compiler to another. The expression `&structure` is always the address of the structure, but `structure` can represent different things. In the older compilers this represents the starting address, like the expression with the `address` operator. If `structure` is passed with a compiler that can already pass the data structure, the entire data field is made available to the calling function (not just the 4 bytes which represent the pointer to it).

20.2 Initialization

This word should also look familiar from earlier chapters. Before you continue, here's a brief reminder of just what initialization does.

This expression designates the first value assignment of a variable. Before using a variable, it must contain a defined value. Otherwise the result of calculations may be nonsense, or the system may crash. The initialization can be a direct assignment in the following form:

```
int i;
i = 0;
```

Or condensed into one line:

```
int i = 0;
```

The initialization in the definition has the advantage of not requiring additional assignments to set the variable. This saves time and memory. Declarations, definitions and initializations can be combined with one data type:

```
double number, pi = 3.1415926, sin();
```

C permits any constants and expressions during the initialization. The following assignments can be found in one line:

```
long number = x * pi - abs(y);
char *cp = string + strlen(string);
```

The variables used were already initialized or `number` would have contained an undefined value.

Braces can improve visibility for arrays and structures, and separate individual entries from each other. A pair of braces must be placed before and after the data which are transferred to the variables. A comma follows the fields, even if the braces were used. After the initialization there is a semicolon which is often omitted, causing compiler errors. Some examples for correct structure definition:

```
struct CAR  {
              char make[16];
              int hp;
              int cylinder;
              double price;
            };

      struct CAR will_have =
```

```
{
  "BMW",
  120,
  4,
  40000.0
};
```

Or collected together through the structure definition:

```
struct CAR  {
              char make[16];
              int hp;
              int cylinder;
              double price;
            } will_have =
{
  "BMW",
  120,
  4,
  40000.0
};
```

The examples of multi-dimensional array initialization were already discussed in the chapter about arrays and pointers. Limitations because of memory classes were also mentioned. Initialization is only permitted for `global`, `external` or `static` variables. If an `auto` variable appears within a function which corresponds to this initialization, e.g.:

```
char message[] = "Remember the Initialization!";
```

This can be fixed with a pointer definition:

```
char *message[] = "Remember the Initialization!";
```

There are no limitations if the lower definition is used as a pointer variable. Another possibility is the use of `static` variables. It doesn't matter whether the string is stored in a `static` or `auto` variable. One small word, but a big difference.

```
static char message[] = "Remember the Initialization!";
```

21.
Pointer Arrays

21. Pointer Arrays

You worked with pointers and arrays in earlier chapters. As the title indicates, they can be combined to construct an array of pointers. You may be wondering what you can do with a pointer array. If you marketed an existing program in a foreign country, you'd have to find every piece of text in the source code and translate the text into that foreign language. It would be simpler and safer to store all the text in one area of the program, or even in a separate file, and let the translator change it from there. A pointer array would point toward that area or file.

Let's start with a list of error messages that the user might see after entering incorrect input. The use of error numbers makes sense, since some errors occur at several different locations. In the current program portion, passing the error number to the error routine is sufficient because the function should do the rest.

One solution to this problem would be a function that tests the occurrence of this error or another error, and display a message if necessary.

```
error(e_number)
int e_number;
{
  switch(e_number)
    {
      case 0:
        puts("Everything OK, no error!");
        break;
      case 1:
        puts("Wrong key activated!");
        break;
      case 2:
        puts("Please insert diskette!");
        break;
      default:
        puts(Unknown error occurred!");
    }
}
```

puts

This is a fairly complex implementation of the function which requires another `puts` call for each additional error message. The `puts` function displays a string on the monitor, without the options available in `printf`. It is a little faster than the general output functions, but various `case` statements must be added, which slow down the program. Since every number can be assigned a certain error message (string), it should be possible to use the error number as an index to a field of strings. Since a string is usually stored in `char error[81]`, the memory for the text is defined as a two-dimensional array:

```
char error[32][81];
```

Now there is space for 32 strings with a maximum length of 81 characters each. This formulation permits the following routine for error message output:

```
error(e_number)
in e_number;
{
     puts(error[e_number]);
}
```

But the work which was saved here, must be completed elsewhere. Each string must be initialized with the `strcpy` function. The following command sequence shows how:

```
strcpy(error[0],"Everything OK, no error!");
strcpy(error[1],"Wrong key pressed!");
strcpy(error[2],"Please insert Diskette!");
...
```

It doesn't matter whether text is called with `strcpy` or `puts`. A disadvantage of this method is that memory gets allocated elsewhere. The definition allocates 81 characters, including a null byte for every error message, even if a message only requires 20 characters. This doesn't matter too much in the Amiga, but the user shouldn't develop bad programming habits. If you do the same thing for a text in which each word is assigned an entry, memory rapidly fills with garbage.

As a last resort, the string arrays can solve almost all the problems mentioned above. The definition of a string array is as follows:

```
char *error[32];
```

The pointer can be set to the beginning of an error message and can make the message length dependent on a fixed array length. The next string starts immediately after the last character of the previous string. This avoids initialization.

You may remember that a program in Chapter 15 pointed to a string within program text. This can occur during pointer definition; the entire pointer array is initialized with the starting addresses of the strings. The error messages must be defined as `global` if this turns out to be the case:

```
/* error_msg.c 21 */
char *error[] =
{
"Everything OK, no error!",
"Wrong key activated!",
"Please insert Diskette!"
};
```

```
main()
{
/* Display all error messages */
int i, error_msg = sizeof(error) / sizeof(char *);

for(i = 0; i < error_msg; i++)
printf("Error Number %d: \"%s\"\n", i, error[i]);
}
```

Since the number of error messages are not counted, they aren't indicated during the definition. Because of this, additional text can be entered between braces without making changes. In the actual program, however, the number must be calculated. The memory requirement of `error` can be obtained from `sizeof`. The `error` function is now an array of pointers. Now `sizeof` reports that the variable consumes 12 bytes. That is the memory requirement for any pointer, and has no bearing on the memory needed for the text. The 12 bytes are divided by the space requirement of a `char` pointer (which is 4 bytes). The result is the number of pointers, the maximum index minus one (The indexes start with 0).

22.
Useful Macros

22. Useful Macros

Much work has already been done with `#define`. The substituted text, called a *macro*, was kept simple; one word exchanged for another. But that's only half the job, macros also allow you to pass arguments. Some useful macros have been developed to make parameter passing easier for the user.

Construction of a macro

Functions that do little and have concise coding can be written as macros. Since macros replace the original text, the compiler translates the C code directly to machine language at that location. Function calls or parameter passing is not required. The required directives are located at the exact location in the program. This makes the macros faster and more efficient than function calls. If used frequently, however, macros make the program code much larger. The same operations are repeatedly stored in identical form at the exact place where they are needed in the program code. An advantage of macros is that they are usually independent of data types. This condition can be seen in the example of the MAX macro created earlier in the book:

```
#define MAX(a,b) ((a>b)? a : b)
```

If variables `i1` and `i2` have been defined as integers, the following macro results:

```
result = MAX(i1,i2);
```

From the preprocessor:

```
result = ((i1>i2)? i1 : i2);
```

The variable `result` is also an integer value. If `i1` and `i2` were defined as `float` values, the same expression occurs, but a `float` number is returned for `result`. In text replacement it doesn't matter which data type was used. This is impossible with functions because the data types are specified for parameters that are passed. This is extremely easy to use.

22.1 Macro Error Sources

Improperly implemented macros harbor some dangers which can lead to errors. These errors can be extremely difficult to detect, but most can be prevented with little effort.

If the macro call just defined included some parameters which contained operators, errors could appear. For example:

```
result = MAX(i1 | 2,i2);
```

is converted as usual into:

```
result = ((i1 | 2>i2)? i1 | 2 : i2);
```

The > comparison operator has a higher precedence than the | character. This means that first a test is made to see whether `i2` is less than 2. The result of this logical comparison (0 if false, 1 if true) is then combined with `i1` using OR, bit by bit. This could not occur in the basic calculations since they have a higher precedence than the comparison operators (see the Appendices for a table of all precedences). A simple remedy is to place all parameters found in the macro inside parentheses. Use this definition:

```
#define MAX(a,b) (((a)>(b))? (a):(b))
```

Side-effects caused by calculations and value changes can also cause problems. A simple example is the following short program which should calculate the squares of numbers between 0 to 10.

```
/*bad_macro.c 22.1 */
#define QUADRAT(x) ((x)*(x))

main()
{
  /* Wrong use of a Macro */
  int i = 0;
  while(i <= 10)
    {
      printf("The square of %d ", i);
      printf("is %d\n", QUADRAT(i++));
    }
}
```

The output of the program is:

```
The square of 0 is 0
The square of 2 is 6
The square of 4 is 20
```

```
The square of 6 is 42
The square of 8 is 72
The square of 10 is 110
```

Where's the mistake? Examine the material left by the preprocessor for the compiler. The line with the macro is the important line:

```
printf("is %d\n", QUADRAT(i++));
```

It becomes:

```
printf("is %d\n",((i++)*(i++)));
```

If the square of 2 (`i=2`) is computed, the following expression is what was actually calculated and passed to the `printf` function:

```
2 * 3
```

The variable is incremented before the multiplication, and the second multiplier is wrong.

You may have wondered why the program uses two `printf` calls instead of making do with one. There is another error source here which must be considered as a separate entity, otherwise this overview would not cover all possible side-effects. To demonstrate the errors which occur in `printf` functions, let's try this with the same function, but without a macro:

```
main()
{
  /* Wrong use of a Macro */
int i = 0;
while(i <= 10)
{
printf("The Square of %d is %d\n", i, i * i++);
}
```

As a result you get two squared numbers which don't match the desired numbers. 4 is offered as 3's square (the square is only calculated for the preceding number). This is caused by the parameters being placed on the function's stack (temporary storage), where the function expects to find them. Unfortunately the storage of these values occur in reverse order, i.e., first `i * i++` is stored, and `i` is incremented. Then the first parameter of `i`, which already has the wrong value, appears.

C language offers many routes to writing short and efficient programs. However, there is a danger of the new programmer trying too much, too soon. The side-effects shouldn't occur in functions and macros unless you know their effects on all variables and parameters. Again, make use of the precedence table as needed (see the Appendices for this table).

22.2 Library Macros

Frequently-used macros are best stored in a library, from which they can easily be inserted in source code with `#include`. This group includes various conversion functions for letters (e.g., testing if a letter is upper or lowercase). The following `defines` have the following tasks:

Convert uppercase letters into lowercase letters:

```
#define to_lower(c) ((c)+32)
```

Convert lowercase letters into uppercase letters:

```
#define to_upper(c) ((c)-32)
```

Test for letters (yes = 1, no = 0):

```
#define isalpha(c) ((c>>='A' && (c)<='Z' || (c)>='a' &&
(c)<='z')
```

Test for uppercase letters (yes = 1, no = 0):

```
#define isupper(c) ((c>>='A' && (c)<='Z')
```

Test for lowercase letters (yes = 1, no = 0):

```
#define islower(c) ((c>>='a' && (c)<='z')
```

Test for number (yes = 1, no = 0):

```
#define isdigit(c) ((c>>='0' && (c)<='9')
```

Test for alphanumeric characters (letters or numbers) (yes = 1, no = 0):

```
#define isalnum(c) isalpha(c) || isdigit(c))
```

Test for blank, tab, linefeed, carriage return, formfeed (yes = 1, no = 0):

```
#define isspace(c) ((c)==' ' || (c)=='\t' ||(c)== '\r' ||
(c)=='\n' || (c)=='\f')
```

Test for special characters (yes = 1, no = 0):

```
#define ispunct(c) ((c)>=' ' &&!isalnum(c))
```

Test for printable characters (yes = 1, no = 0):

```
#define isprint(c) ((c)>=040 && (c)<=0176)
```

Test for control characters (yes = 1, no = 0):

```
#define iscntrl(c) ((c)>=0 && ((c)==0177 || (c)<' '))
```

Test for ASCII characters (yes = 1, no = 0):

```
#define isascii(c) ((c)>=0 && (c)<0200)
```

These `defines` should be easily understood once they are examined. They should be written into a file named `CTYPE.H`, unless this type of file is already available in a subdirectory. If the following line occurs in a program, you know what should be found there:

```
#include <ctype.h>
```

Remember that for the test for letters, only the 26 letters of the alphabet are considered. International special characters are not viewed as letters. Maybe they can be implemented in a suitable manner. Perhaps the `strcmp` can be converted with the new `defines`.

```
char *s, *t;
int n, compare;

compare = strcmp(s, t);
compare = strncmp(s, t, n);
compare = stricmp(s, t);
compare = strnicmp(s, t n);
```

The first function is identical to the routine you programmed. It compares two strings and returns the result of the comparisons. In the second function `strncmp`, the third value indicates up to what point the comparison should be made. The comparison can be limited this way to `n` characters. For example only the first four elements. The functions which have an `i` in the name, don't differentiate between upper and lowercase letters. Comparing the two strings "aBcDEf" and "ABcdeF" with the function returns the value zero because both strings are equal.

23.
Communication

23. Communication

The programs written so far have only displayed data on the screen or requested keyboard input. It's time to communicate with other devices. The CLI is the easiest way to transfer data. In this chapter, we'll communicate with the CLI as well as other devices.

23.1 Passing Data with the CLI

All programs are called from the CLI by entering the filename and arguments (if needed). Here is one type of call to invoke ED:

```
ED file.c SIZE 50000
```

This complete line can be made available to the called program, though not in this form. The operating system modifies this line slightly.

Now comes the question of data transfer. The main function containing two arguments controls this. Until now every call appeared as follows:

```
main()
{
....
}
```

The next code passes two values from the calling program from either the CLI or a MAKE file. The first value represents the amount of information and a pointer to a char pointer. This sounds somewhat complicated, but looking at the input line of the program should make it clear. First the new version of main with two arguments:

```
main(argc, argv)
int argc;
char *argv[];
{
  . . . .
}
```

The name of our fictional program is prg. Look at this sample entry:

```
prg Text1 parameter 3 -pi
```

After the program call, the variable `argc` contains the number of arguments (5). Why is 5 passed when only 4 arguments are available? The fifth argument comes from adding the program name used during the call. The name `argc` (ARGument Count) is a random choice, since it is an `auto` variable of the `main` function. The name `argv` (ARGument Vector) handles vectors.

Spaces or tabs separate every argument of the input line. After the call the pointers of `*argv[]` point to:

```
argv[0] "prg"
argv[1] "Text1"
argv[2] "Parameter"
argv[3] "3"
argv[4] "-pi"
```

Let's examine the data. Use the following program to print the data:

```
/* arg_test.c 23.1 */
main(argc, argv)
int argc;
char *argv[];
{
  while(--argc >= 0)
    puts(*argv++);
}
```

What can be done with this? You can access a small math program by entering the following in the `CLI`:

```
compute 123.5 * 4711
```

The following program is written so that it will only perform simple calculations consisting of two numbers and an operator. Feel free to improve on the program as needed.

```
/* arg_math.c 23.1 */
extern double atof();  /* Declaration */

int error = 0;


main(argc, argv)
int argc;
char *argv[];
{
   double result, value();

   if(argc != 4)
     printf("\nWrong Entry\nCall: number1 # number2\n");
   else
     {
       result = value(argv[1], argv[2], argv[3]);
       if(!error)
```

```
     printf("\n%s %s %s = %.9lf\n", argv[1], argv[2],
argv[3], result);
     }
}

double value(number1, op, number2)
char *number1, *op, *number2;
{
    double z1 = atof(number1);
    double z2 = atof(number2);

    switch(*op)    /* only the first character */
      {
        case '/':
          return(z1 / z2);
        case '*':
          return(z1 * z2);
        case '-':
          return(z1 - z2);
        case '+':
          return(z1 + z2);
        default:
          printf("\nUnknown Operator >%s<\n", 7, op);
          error = 1;
          return(0.0);
      }
}
```

Lattice

The mathematical function/floating point library must be linked to the standard library because it contains the `atof` function. Example:

```
lc -Lm  math2
```

Aztec

If you work with Aztec C, the mathematical function/floating point library must be linked to the standard library `C.LIB` since it contains the `atof` function. Example:

```
cc +L math2.c
ln math2.o -lm -lc
```

23.2 Buffered Input/Output

Many programs require permanent data storage for files, whether it is a database or a word processor. These require routines that control input/output with external devices such as printers, disk drives, RS-232 interfaces or a hard disk drive. The operating system provides various functions for this purpose. These routines can be divided into two groups; buffered input/output and unbuffered input/output.

This type of data transfer does a lot of work, even though it may not seem evident at first. For example, all data selected for transfer to disk goes to a buffer first. When this buffer completely fills, the data goes to the disk. The question is, why do it this way?

A disk drive reads and writes information much slower than the computer can send or receive it. This is caused by the mechanics of the disk drive. Before writing any data, the read/write head must move to the track where the data is stored. Then it must wait until the disk rotates to the right location. Only then can data be written. Although this timing is brief in human terms, the computer (actually its central processing unit or CPU) is kept waiting a very long time.

Disk buffers

If you transmitted every character with this method, the computer would spend more time waiting to place a single character on the disk drive than performing any other task. For this reason, smaller amounts of data move to an area of memory in the computer before transmission. Once this buffer fills, the data moves to the disk drive. This reduces computer waiting time. As soon as the drive writes the first character to the proper place, it can place the other data right behind it and write the complete buffer in one pass. This reduces the number of disk accesses, which accelerates program execution. The same principle is also used for reading data.

The functions which perform this job are `getc` and `putc`, which like their relatives `getchar` and `putchar`, input or output a character.

These functions are not part of the C language. For this reason they can be found in a library, or in this case as a `define` in a header file (`.h`). The definition of `getc` and `putc` can also be found in the `stdio.h` file with the familiar `putchar` and `getchar`.

```
#define getc(p)  (--(p)-> _rcnt>=0? *(p)->_ptr++:_filbf(p))
#define getchar() getc(stdin)
#define putc(p)  (--(p)-> _wcnt>=0? *(p)->_ptr++=(c)):_flsbf((c),p))
#define putchar() putc(c,stdout)
```

Let's clear some of this up. The definitions of `putchar` and `getchar` are all you need to know for now. Both can be traced back to `getc` and `putc` and represent special versions of the two functions.

Before the first character can be moved with these functions, a channel must be opened. A channel is just a data line to a certain device. No cables actually open up a channel, but the system knows where to send the information. A special code obtained from the operating system during the opening of the channel allows addressing the device at any time. Devices and individual disk files can be addressed.

Several files can be addressed on the same drive without having data conflict. The file pointer indicates the channel. Since a buffer is used for input/output, the computer must be informed of where the data should be stored intermittently and how large a space must be reserved. A structure named `FILE` (notice the uppercase letters) defined in `stdio.h` contains all necessary data for buffered input/output. The following listing writes a file and then reads it again:

```
/*fprint-fscan.c 23.2.1 */
#include <stdio.h>

main()
{
   FILE *input, *output, *fopen();
   char filename[81], text[200];

   printf("Please input file name!\n");
   scanf("%80s", filename);
   printf("Input a (long) word!\n");
   scanf("%200s", text);

   output = fopen(filename, "w");
   printf("Filehandle %d\n", output);
   fprintf(output, "%s", text);
   fclose(output);

   input = fopen(filename, "r");
   printf("Filehandle %d\n", input);
   fscanf(input, "%200s", text);
   fclose(input);
   printf("The Text: >%s<\n", text);
}
```

The `fopen` function

The `fopen` function opens the channel and returns the file pointer, used for all future access, to this file. Since `fopen` returns something other than an integer, it must be declared as a function which returns a file pointer. This routine requires two arguments: the name of the file and the access mode. The user can enter the name. The access mode tells the computer what should be done with the file. The mode can be one of three letters:

r (read)	opens a file for reading
w (write)	opens a file for writing
a (append)	opens a file for adding additional data

A file opened for reading (r) can only read data, not write data. A file open for writing (w) lets you write data. The append (a) mode writes data to the end of an existing file. In normal write (w) mode, writing starts at the beginning of the file and overwrites existing data. This can easily lead to loss of data. The example above overwrites an existing file and destroys some previously stored data.

After the opening, the file is ready for writing. The file handle appears on the screen. The `scanf` function assigns the characters entered through the keyboard to the string `text`. The `fprintf` function can write data to a file. It is almost identical to the `printf` routine, but differs in the first argument. Before the command string, a file handle must be passed to assign the information to the correct file. After writing, the file closes. This step is very important because of the buffer. All input/output goes there for intermediate storage until the buffer is filled. Some of the data input can still be stored in that buffer. If the user assumes that everything was stored on the disk and switches off the computer, the data still in the buffer would be lost. For this reason the `fclose` call closes the channel after writing the remaining buffer contents to the open file.

Now the file reopens again, but this time for reading which is signaled with mode (r). Since keyboard input always uses `scanf`, the `fscanf` is used here. First the file pointer and the arguments of the `scanf` routine must be passed to the `read` function.

The correct closing of the file follows. If you omitted this instruction, data loss cannot result. However, it's good practice to close an opened file immediately after use, not only because it is good housekeeping, but also because a computer can maintain only a certain number of open files. If more files are opened, a channel cannot remain open at a certain time. Should an error occur, because the operating system cannot make a channel available, the file pointer is returned as zero. This happens when no channels are available, or the file which should be read does not exist.

The next example program is a small copy program. It is called with arguments and therefore is able to accept arguments from `main`.
One difficulty must be avoided. Nobody knows in advance what type of data will be transmitted. The `fscanf` cannot be used since it has to indicate if strings or numbers are used. The program can only read one character at a time. The `scanf` routine with the format instruction `%c` can be used, but the `fgetc` function works much better. It reads a character from an input file and is much faster than the `fscanf` function. The `fputc` statement performs the output.

After opening the two files, a character is read and displayed immediately, until.... You don't know when all data has been copied. How can you detect when the last character has been read? The problem has already been solved. `fgetc` returns a special character if no additional information is available—end of file (`EOF`). The `#include` file `stdio.h` contains this text as a `define` so that the incoming characters only have to be compared with `EOF`.

`EOF` is stored there as -1. This has consequences which at first are not evident. Valid data have codes which in `fgetc` are between 0 and 255. When -1 appears, no `char` variable can be selected to accept the character. Either negative numbers are ignored or data is lost. For this reason, `int` variables are used even if only a `char` element is stored in them.

```
/* copier.c 23.2.1 */
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
       long copy(); /*If it is interesting */

       if(argc !=3)
          {
              printf("Bad Arguments!\n");
              printf("From_file to_file\n");
          }
       else
         copy(argv[1], argv[2]);
}


copy(fromfile, tofile) /* Copy Routine */
char *fromfile, *tofile;
{
       FILE *input, *output, *fopen();
       register long counter = 0;
       register int c;

       if(!(input = fopen(fromfile, "rb")))
        /* Open as Binary        file */
          {
            printf("%s can not be opened!\n", fromfile);
            return 0L;
          }
       if(!(output = fopen(tofile, "wb")))
          {
            printf("%s can not be opened!\n", tofile);
            fclose(input); /* Was OK */
            return 0L; /* Zero */
          }

       while( (c = fgetc(input)) !=EOF)
        {
```

```
            fputc( c, output);
            counter++;
          }

        fclose(input);
        fclose(output);
        printf("\n%ld Bytes copied!\n", counter);
        return(counter);
}
```

This copy program is called as follows:

```
copier [d:] [\path]name1[.ext] [d.][\path]name2.[ext]
```

Everything written into the brackets is optional and can be omitted. Only two filenames must be provided. If an error should occur during the opening of the two files, an error message appears. An error message is also displayed if too many or too few arguments are passed.

Since it is very slow, this program is unsuitable for everyday use. However, it's a suitable demonstration program. Let's limit the discussion to the essentials. The files are opened as binary files with `rb` and `wb` (Aztec compiler users should omit the `b`; the file always opens as a binary file). For example, this prevents any conversions being attempted which could occur in text files. During reading all `\r` characters (carriage return) are erased automatically and characters with code 26 (<Ctrl><Z>) are converted to `EOF`. During `write` the linefeed (`\n`) is converted to a character combination (`\r\n`). Opening the file with the appendix `b` (binary) returns all characters as they are stored in the file. During the `write` all characters sent are stored in the file and not converted.

23.3 More Buffered Input/Output

Besides `fgetc`, `fputc`, `fscanf` and `fprintf` there are some other important functions that use the internal buffer. Among them are `fread` and `fwrite`. These routines transport any number of bytes. For this reason two additional arguments are required for `fread` and `fwrite`. One argument is the area which serves as the buffer and the other the size of the units to be transmitted. This needs some explanation. The buffer in previously used functions was always located in the `FILE` structure. Since only small amounts of data were transported, the buffer did not have to be large (512 bytes). Since the user can now determine how much data is transmitted, this buffer may be too small. For this reason the user must define the memory area, thereby setting the maximum size of the data transfer. The data size must also be indicated. In `getc` and `putc` only one character can be transmitted (1 byte) and the size of the `char` object doesn't have to be indicated. If a `long` value, instead of a character, is stored, 40 bytes must be transmitted for 10 of these values. The object size (in this case four bytes each) is the second argument that must be passed. If the data type is unknown, the `sizeof` operator should be used because it returns the correct value.

Besides these two arguments, the number of units to be transmitted (`char`, `int`, `structure`, ...) and the file pointer must be transmitted. The buffer size determined at the time of definition should not be too small. A call of this function appears as follows:

```
Datatype buffer[Element]; /* Definition of the buffer */

fread(buffer, sizeof(Datatype), Element, file_ptr);
```

The first argument is the buffer from which the data is read. The buffer should be the same type as the units to be transmitted. The second value is the unit size. This unit is a data package which can be transmitted as one item. If `long` variables are transmitted, it makes sense to indicate 4 bytes as data block length, since 4 byte units are the normal memory requirement. If 100 `double` variables are stored, 100 is placed at the element. Finally a file pointer is added, which was received from `fopen`.

If data is read or written, the argument sequence and type remains the same. The `fwrite` function stores the data using this syntax:

```
int table[876], size = 876;
FILE *outpt_ptr;

fwrite(table, sizeof(double), size, outpt_ptr);
```

If the value of size is not specified, either through a variable or with a `define`, the `sizeof` operator can be read:

```
sizeof(table)/sizeof(int)
```

The function returns the number of completely transmitted data packages. This ensures storage of all data. During copying data can be read until the number of requested data differs from the data delivered. If `fread` returns a zero, the last data was read. A sample program:

```
/* fread.c 23.3 */
#include<stdio.h>

#define NUM_DATA (sizeof(data)/sizeof(long))

Long data[] =
{
  4711, 815, 1024, 1, 31415926, 0, -13, 10,
  0xFFFF, 065432
};

main()
{
  FILE *input, *output, *fopen();
  int i;
  long test[NUM_DATA];
  char filename[81];

  printf("Please input filename!\n");
  scanf("%80s", filename);
  printf("Data size %d, Elements %d\n",
        sizeof(data), NUM_DATA);
  output = fopen(filename, "wb"); /* binary! */
  fwrite(data, sizeof(long), NUM_DATA, output);
  fclose(output);

  printf("Read Data!\n");
  input = fopen(filename, "rb");
  printf("%d Elements read\n",
       fread(test, sizeof(long), NUM_DATA, input));
  fclose(input);
  for(i = 0; i < NUM_DATA; i++)
    printf("%ld\t", test[i]);
  printf("\nDone!\n");
}
```

This program writes a long array to the file after entering the filename. The array size and the number of elements appear on the screen. The `define` NUM_DATA stores all data on the disk. After closing the file, another array accepts the data read. All data appears on the monitor.

Lattice C users must open the file as binary, or internal conversion produces false values in the variables. If `fwrite` and `fread` are used, the file must also be opened as a binary file.

23.4 Unbuffered Input/Output

Besides the buffered functions just demonstrated, there are other routines which do not require a buffer. The data to be transmitted don't have to be stored in a buffer, but can be stored immediately. This cancels all the effort required for the buffer and internal pointers. This also cancels the need for a file pointer through which the operating system can access the buffer. A channel number assigned during the opening must be used as identification. This channel number is stored in an integer variable and replaces the file pointer in all calls. The `open` function opens a file. Filename and file mode (as an integer) arguments pass to the function. In `fopen` the mode must be a string, while in `open` the mode is one of three values: 0, 1, 2 or 8. These numbers correspond to the strings r, w and a.

0	opens a file for reading
1	opens a file for writing
2	opens a file for reading and writing
8	opens a file for appending

Instead of these numbers `defines` can be used to make the program more readable. They are stored in a header file named `fcntl.h` and are defined as follows:

```
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR 2
#define O_APPEND 8
```

To use these defines, the file must be included in the source with the sequence:

```
#include <fcntl.h>
```

Another difference from the `fopen` routine is that `open` always assumes the existence of a file. The following call creates a file even if the file already exists:

```
fopen("filename", "w")
```

The `open` function always prompts for the name of an existing file. The `create` function must be used to create a new file.

The `create` function returns an integer (the file handle). You don't need to call `open`. If an error occurs and the file cannot be opened, both `create` and `open` return the value -1. Before using the returned value as a file handle, check the file handle for a value of -1, or the system

will crash. The key combination <Ctrl><Commodore><Amiga> resets the computer, but deletes any data in the RAM disk.

A file opened in this manner allows writing using the `write` function instead of the `fwrite` function. Since this method uses no buffer, only certain input/output functions can be accessed. Also, the `read` function replaces the `fread` function. Buffered functions have an `f` in front of their name (e.g., `fopen`, `fread`, `fclose`). Unbuffered functions omit the `f`. The `close` function closes an unbuffered file.

The following function stores a list of `double` numbers.

```
/* write-read.c 23.4 */
#define NUMBER (sizeof(data)/sizeof(double))

double data[] =
{
  1.5, 2.0, 3.14159265, 2.718281828,
};

main()
{
   int handle, dummy = 0, i, actual;
   double data2[NUMBER];
   char filename[81];

   printf("Please input filename!\n");
   scanf("%80s", filename);

   handle = creat(filename, dummy); /* Create new */
   if(handle != -1) /* Everything OK? */
     {
       actual = write(handle, data, sizeof(data));
      printf("Desired %d Bytes, Actual %d Bytes\n",
           sizeof(data), actual);
      close(handle);
     }
   else
     printf("Error during creation of %s\n", filename);

   handle = open(filename, 0, dummy); /* Read */
   if(handle != -1) /* Everything OK? */
     {
       actual = read(handle, data2, sizeof(data));
       printf("Desired %d Bytes, Actual %d Bytes\n",
          sizeof(data), actual);
       close(handle);
     }
   else
       printf("Error during opening of %s\n", filename);
   for(i=0; i < NUMBER; i++)
      printf("%.8lf  ", data2[i]);
   printf("\nThat's all!\n");
}
```

The `open` and `create` functions, which return a file handle, have a peculiar variable named `dummy`. This variable represents a value which may be unnecessary, but the compiler checks for this variable if needed. The value stored in `dummy`, as the name suggests, has no significance. In `open` the second argument represents the `dummy` value for reading or writing.

The `write` and `read` functions have one less argument than `fwrite` and `fread`. Furthermore (and this is important), the file handle is placed at the beginning, not at the end as in the buffered functions. The unbuffered functions transport the data one byte at a time only. This means that the size indication is unnecessary. The return value is the number of bytes transmitted so far.

23.5 Direct Access

The following functions provide the user with the ability to directly access certain characters in a file. The difference from the usual `read` lies in the fact that not every character must be read starting at the beginning of the file until the program finds the particular characters. To access the last ten characters in a file of 1,000 characters, 990 characters would have to be read first. With direct access, the command starts the reading after the 990th character. A file pointer always points to the last accessed data. During a sequential `read` or `write`, when one character after another is processed, this pointer always increments by one. The functions `lseek` and `fseek` let the user set this pointer to any desired position. Both routines require three arguments for this, where `lseek` is the unbuffered version and `fseek` the buffered version. For this reason `lseek` requires the file handle as the first argument, while `fseek` expects a file pointer. A second value follows the number of bytes by which the pointer must be moved. Positive values move the pointer toward the end of the file, negative values toward the beginning of the file. This value must be passed as a `long` value. The third argument, an integer, indicates from which position the movement should start. A 0 sets the data pointer to the beginning of the file, 2 to the end of the file and 1 to the current position. Some examples:

```
lseek(f_handle, 100L, 0);
```

The data pointer moves to position 100 (i.e., 100 characters from the start of the file). It now points to the 101st byte of the file. The following function places the pointer at position 70, since the call passed the value 1:

```
lseek(f_handle, 70L, 1);
```

The 1 indicates the calculation of the new pointer position from the current location. To move the pointer 20 characters toward the beginning of the file, the following function is required:

```
lseek(f_handle, -20L, 1);
```

If the processing of a file should start from the end of the file, the pointer can be set to the last position of the file with:

```
lseek(f_handle, 0L, 2);
```

This shows that in mode 2 (end of the file), only negative numbers or 0 are permitted since the pointer is already at the end of the file. In mode 0, only positive numbers or 0 can be used. These functions only move

the file pointer. The data must be read or written with the various functions such as `read` or `write`.

The `lseek` function returns the value of the data pointer after the move. `fseek` returns either 0 or -1. With -1 an error occurred, otherwise everything proceeded without a problem.

Two additional routines, a buffered and an unbuffered version, can sense the current value of the data pointer. Since it is a `long` value, the `ftell` and `tell` functions must be declared first. The only required argument of both functions is the proper file handle.

The function call could be replaced with the call:

```
lseek(f_handle, 0L, 1);
```

which indicates the value of the current data pointer.

23.6 Reading a Character

Our previous programs used `scanf` to read a character from the keyboard. The problem with this is that you have to press the <Return> key after every character. A word processing program would be intolerable under these conditions. Even a modest application such as controlling the cursor in all four directions would be difficult.

The `getchar` function

The `getchar` function offers help. This function receives the pressed key's code immediately. Even here there is a difference between theory and practice. Almost all other C implementations use this function according to rules—except for the Amiga. The Amiga requires the <Return> key for execution.

23.6.1 Standard Input/Output

Usually data is entered through the keyboard into the computer. Messages and the results of calculations usually appear on the screen. These two devices combined are called the console. If you do not instruct the computer to get the source data from or send the destination data to a particular device, it defaults to the standard input/output (keyboard and screen). The good news is that these devices can be changed by the user. The Amiga can input data from another device instead of the keyboard.

Standard devices

The standard input device is the keyboard. The standard output device is the monitor screen, or screen for short. In various windows the output defaults to the `CLI` window. This window is the standard output for your previous programs. All `CLI` limitations also apply to your program. This causes the error in `getchar`.

The `CLI` is line-oriented. Data processes after you enter your input and press the <Return> key. Anything can be typed in without the computer reporting, during input, that this isn't permitted. Press <Ctrl><G> to make the screen blink. Even though these characters can be entered, you cannot display them on the screen. If you try this, the blinking function executes. The `CLI` is an input console which processes whole lines and not single keys.

This rule of using only complete lines also applies to programs started from the `CLI`. Now that you know why <Return> must be pressed, it's time to create a user-defined window to solve this problem.

23.7 A User Window

Creating a personal window is fairly simple. Developing your own window consists of opening an output file, since the output is written into this newly created window. The keyboard then acts as the input device, according to the arguments set by the user window.

23.7.1 The Three Windows

You have three device options for opening non-CLI windows:

```
*
CON:
RAW:
```

These strings follow the open command in the CLI. They replace the usual filenames and drive specifiers. The asterisk sends data directly to the CLI window; a new window isn't created. Nothing has changed for the input either. Opening an asterisk device only causes the same trouble as before.

The CON: device creates a user window. The open call as listed below creates an unbuffered open file and a pathname of CON:

```
open("CON:0/0/200/50/Title line", 0, dummy)
```

The CON: device name replaces the drive specifier, and the new window's coordinates follow. Finally the name Title line appears in the upper left corner of the title bar. A slash character (/) separates the window arguments. The other arguments such as 0 (read) and the dummy value follow the usual syntax of the open routine.

The returned file handle appears in the examples with all read calls. An 81-character string acts as a buffer. Try this short routine:

```
/*window1.c 23.7.1 */
#define c *character
#define ESC 27

main()
{
   int dummy = 0, num, handle;
   char character[81], line[256];
```

```
    handle = open("CON:0/0/200/50/My Program", 0, dummy);
    printf("Opens %d handle\n", handle);
    if(handle != -1)
     {
       do
          {
            num = read(handle, character, 1);
            printf("Character >%c< Code %d\n", c, c);
          } while (c !=ESC);
        close(handle);
     }
}
```

If you forget to test for a successful opening of a file, the computer may crash using -1 as handle value. When the program executes, a new window appears; nothing happens after you press a key. All keys are processed only after you press the <Return> key. A new window appears. Unfortunately the window only works in line mode, like the normal CLI window. Press <Esc><Return> to exit the program.

The RAW: option also creates a window. The difference between this option and CON: is the way the information is displayed. A RAW: window displays information in "unfiltered" form (i.e., control characters and garbage appear). Change CON: to RAW: in the above example. Compile, link and run this new version and watch what happens next.

Again a new user window appears. You can size this window and move it around the screen. Select the new window and send data to it (type on the keyboard). Notice that the window reacts to every keypress. However, the input doesn't appear in the user window. The program seems to be ignoring the RAW: specification.

All output with `printf` continues to go to the CLI window. This isn't surprising since it's the standard output device for programs started from the CLI. To write something into the user window, for which a special handle has been issued, a `write` routine must be used. Instead of `printf`, use `fprint` which has the same functions.

Wait a minute. The `fprintf` function is a routine for buffered files and an unbuffered file was opened with `open`. It only works if the file handle was obtained with `fopen`. Since the handle is an integer value and not a file pointer, the handle doesn't fit the `fprintf`. However, the library contains a command named `sprintf`. Instead of writing the prepared data into the standard output of the buffer, everything goes to a string. The following example invokes this function:

```
char string[200]; /* not too small */
int test = 4711;

sprintf(string, "The result is %5d\n", test);
```

Now the user can write anything into his own window—the `printf` function isn't required for output into the `CLI` window. Now everything needed for input/output in the new window is available. Here is the corrected listing:

```
/* window3.c 23.7 */
#define ESC 27

main()
{
  int dummy = 0, num, handle;
  char c, line[256];

  handle = open("RAW:50/50/200/60/My Program", 0, dummy);
  printf("Handle %d open\n", handle);
  if(handle != -1)
   {
     do
      {
      num = read(handle, &c, 1);
     /* write(handle, &c, 1); output only the character*/
      sprintf(line, "Character >%c< Code %d\n", c, c);
      write(handle, line, strlen(line));
      } while (c !=ESC);
    close(handle);
   }
}
```

23.8 Redirection

The operating system controls the redirection of data so the programmer doesn't have to worry about it. The use of standard input/output acts as the condition for ensuring data redirection. This includes familiar functions such as `printf`, `scanf`, `putchar`, `getchar`, `puts`, etc.

Look again at the first `RAW` program above which used `printf` for text output. This example will help you understand redirection.

The "normal" call for programs not expecting arguments is:

```
program_name
```

Instructions that are executed by the operating system and not by the program can follow the filename. The greater than (>) or less than (<) characters precede these instructions. The characters inform the operating system that the standard input/output should be modified. Here's a practical example:

```
prg <file1 >file2
```

The program never sees these two arguments. The operating system reads the standard input from `file1` and sends the output to `file2`. The operating system also opens and closes the files automatically. The greater than (>) and less than (<) characters indicate the direction of data, as an arrow would indicate direction. You can immediately see that the data goes to the filename `file2`. Let's examine this process using the `window2.c` `RAW` program in the previous section. The `printf` can be redirected so that all text goes to a file or a printer (device `PRT:`). Start the program (called `window2` here) with the following line:

```
window2 > output.data
```

The new window appears again but no keypresses seem to affect the window. That's all right, since the output which would otherwise appear in the `CLI` window now goes to the file `output.data`. After typing on the keyboard for a while, press the <Esc> key. The window disappears and the user returns to the `CLI`. The `output.data` file can now be read using the `TYPE` or `ED` commands.

Input can also be redirected from the keyboard to a file. This is how a `MAKE` file could be created. In these programming examples this wouldn't make sense since you don't read from the standard input. A redirection would not make much sense in this case.

The standard input/output is a normal line. Just as in opening a file, a file handle is returned. Since these lines are always open, the programmer doesn't have to worry about them. There are of course variables for these handles. They are:

```
stdin
stdout
stderror
```

The `stdin` variable represents "standard input" and the `stdout` variable represents "standard output." What is the third?

In addition to the input and output of "normal" information, C offers an error channel. This makes sense in the following situation.

As described above, the information is redirected. Because an error occurred in the input (e.g., a nonexistent input file) an error message appears on the screen. Wait, the standard output was written to another file. This would mean that the error message was written to the file and the user might not know about this error. For this reason error messages use a separate channel to display user messages.

All three variables (`stdin`, `stdout` and `stderr`) can be used as file pointers which are returned through `fopen`.

Caution: These buffered input/output routines should not be confused with the unbuffered ones. Permitted functions with the file pointers above would be `fprintf`, `fwrite`, `fread`, etc.

The definition of `getchar` can be found in the file `stdio.h` under:

```
getc(stdin)
```

24.
Tricks and Tips

24. Tricks and Tips

During programming unexpected errors can sometimes occur. For a novice programmer, the reasons for these problems can be very hard to find. Much work is often required to determine whether the cause of the problem is in the source code, the compiler or the operating system. This chapter has a few hints for helping you find those errors.

You'll find that this chapter contains a number of tips and tricks for C programming on the Amiga. These tips include the creation of C programs that are accessible from the Workbench, preprocessor directives and macros.

24.1 Starting from the Workbench

You may have already tried to start your own C program from the Workbench. Perhaps you wondered why, despite a full disk, nothing appeared in that drawer's window. The reason for this is that every visible program has an additional file used for storing the program's information and icon data. Every program which appears in a Workbench window has a file with the extension of `.info`.

Our C programs also need `.info` files before they can be accessed from the Workbench. Select a suitable icon on the Workbench (the clock or Notepad, for example). Any other icon can be selected for this program. Now enter the `CLI`. Copy only the source `.info` file to an `.info` file for your file. Remember to use the `.info` extension for both files in this command. The example below copies the `Notepad.info` file to a new `.info` file for the file `test-workb`:

```
copy notepad.info test-workb.info
```

Change `test-workb` to your own filename, and remember to keep the `.info` extension on both filenames.

Quit the `CLI`. Click the disk icon containing the target file and `.info` file to display the icon. If the icon is covering another icon, move it to a free location in the window. Click once on the icon. Press a <Shift> key and click on a disk icon. Select `Snapshot` from the `Special` menu to save the new position.

The following program will tell you how it was accessed, from the Workbench or from the `CLI`.

```
/*access.c 24.1 */
#include "stdio.h"

main(argc, argv)
int argc;
char *argv[];
{ int dummy = 0, handle;
 char line[256];

 if(argc)  /*Argument number not equal to 0 */
   {
  handle = open("RAW:50/50/200/60/My Program", 0, dummy);
      if (handle != -1)  /* No Error on opening */
       {
        sprintf(line, "Started from CLI!\n");
        write(handle, line, strlen(line));
        while(--argc >=0)
          {
            write(handle, *argv, strlen(*argv));
             argv++;
          }
        read(handle, line, 1);  /* Wait for key press */
        close(handle);
       }
  else
   fprintf(stderr, "\nError in opening Window\n");
   }
 else
   {
   printf("Started from the Workbench!\nRETURN-Key!\n");
   getchar();
   }
}
```

It is important that the programmer know whether the program started from the Workbench or CLI. The argument counter `arg` senses this. If `arg=0`, the user started the program from the Workbench. Consider the possible values in the CLI. As a minimum, `argc` contains a 1 only when the program name was used during the call without additional arguments. Otherwise this variable is incremented by the number of parameters. The 0 is an ideal method of differentiation between the two calls. The big question: "Who cares where the program started?"

Start the program `window2`, from the previous chapter, which opens a window. The Workbench creates an additional, useless window. One window is used, the other remains empty. There must be an option of sensing a program start from the Workbench or from the CLI.

In the first case you get a window automatically and the standard input/output is automatically directed to this window. The basic functions such as `scanf` and `printf` can be used, and you can still enjoy the use of your own window. In the second case, the user must handle his own window and input/output.

24.2 Other Preprocessor Directives

The directives `#define` and `#include` should be familiar to you by now. Preprocessor directives make the programmer's work easier. The most important are the following:

```
#undef MACRO
```

Undefining

This is the opposite of `#define`. This undefines (cancels) the macro definition. Assume that two `#define` directives had been used as follows:

```
#define EOS 0
...
#define EOS '\0'
```

Any subsequent program sections use the most recent definition of `EOS`. The first assignment is ignored for the moment. This process can be compared with local variables which use the same name. Access can only occur to the last variable (definition) defined. If the definition is reversed with `#undef`, for example:

```
#undef EOS
```

the defined `EOS` is still present, but now with the first assignment (0).

Partial compilation

The following directive permits the compilation of certain portions of a file, depending on the macro that was defined:

```
#ifdef MACRO
```

Should the macro be defined, the following portion is processed by one of these two preprocessor directives:

```
#elseif MACRO
#endif MACRO
```

If the macro is unknown at this point, all lines are skipped up to the following directives. If an `#elseif` appears the subsequent source code is compiled up to the `#endif`.

These directives are comparable to the C commands:

```
if()
  {
. . . .
  }
else
```

```
  {
....
  }
```

The commands themselves have nothing to do with the final code. The reversal of the process makes the following line possible:

```
#ifndef MACRO
```

Here the macro cannot be defined, so the part following can be compiled. Certain parts of a file can be included or left out through the setting of a `define`, without changing the file to a great extent. Where are the directives used?

In programs intended for compilation on other compilers or even other computers, some differences must be considered such as compiler errors. To achieve an error free compiler run, a `define` can indicate the compiler type. These directives can be found in some header files which come with the compilers.

24.3 Finding and Removing Errors

No program, in the initial stages of development, is free of either syntax or logical errors. The compiler calls these syntactical problems to your attention. Removing these errors shouldn't be a problem. This section describes a few of the possible meanings behind these errors.

- Missing semicolon

If the semicolon is not missing in the line indicated, check one or two lines above the line stated by the compiler and check the level of parentheses.

- Braces do not agree

This error often requires a search of the whole area preceding the error. If you omitted a brace somewhere, the following part is added to the function described up to this point. This in turn leads to mysterious error messages, for example the missing semicolon in the next function definition.

- Wrong answers

If a function which executes flawlessly suddenly returns wrong results, the error may be caused by a failure to declare the function (not for `int`). For this reason all functions not having a return value of `int` should be declared globally at the beginning of the file. The `global` declaration keeps newly added functions, which also use this routine, from erroring out. Complex formulas which work with various operators and data types should generally be in parentheses. This not only makes them more visible, but also prevents errors. Nobody knows all the priorities of various operators by heart and this can quickly lead to problems if a mistake is made. It is better to use one parenthesis too many than one too few.

- Easily confused character combinations

Even experienced programmers have misread these characters:

== and =

&& and &

|| and |

Watch for the legal maximum values of the various data types in calculations. They may differ from one compiler to another:

```
char -128 to 127
int -32,767 to 32,766
long -2,147,483,648 to 2,147,483,647
float -10^38 to 10^38
double -10^303 to 10^303
```

These can never be exceeded, even in intermediate results of a long equation such as:

```
i = (x * y * z - z) / y - x;
```

This can happen quickly with `x * y * z` if the three factors contain large numbers.

- System crash

This may have many causes. For example:

Pointer not initialized

Wrong parameter passed (wrong data types)

Pointer access to odd address (except for char pointer)

25. System Programming

25. System Programming

One of the reasons people work with C language is because of its speed. The main reason Amiga owners use C is probably because the Amiga's operating system was written in C. Knowing the peculiarities of C often helps you understand the Amiga's operating system. This is especially noticeable during system programming when you try to get more performance from the computer.

Intuition

This chapter takes you into the world of *Intuition*. Intuition is the section of the operating system that is concerned with windows, screens, icons, gadgets, menus and the mouse. Intuition's capabilities are so vast that we can explore only a small portion of this material. You'll find additional references suggested if you wish to explore further. We chose to limit ourselves to introductory material on Intuition, especially the creation of windows and screens.

One small warning in advance: Intuition is very complex. You'll see this from the C source that follows in this chapter—the source codes are much more complicated than the ones you've seen so far. Even if you don't understand everything, you should try the programs and experiment with them.

25.1 The Intuition Principle

To use Intuition in user programs, some conventions must be followed. For example, Intuition routines can be used only when the Intuition library has been opened. Intuition is simply a large library containing the functions used in connection with windows and similar things. The difference from the C libraries presented up to now is that these routines are not linked during linking. The user relies on the fact that they are stored somewhere in the computer and are available during program execution. This has some advantages. Since many programs use the same Intuition routines, they don't have to be stored several times in the memory. Every program can use the routines for its own purposes, even if they work with other programs in memory (multitasking). This saves working memory. It makes no difference where these routines are stored in the memory. When you open the `intuition.library`, a pointer is returned to the beginning of this function list. A trademark of Intuition programming is a lot of pointers and structures.

25.2 A Window under Intuition

We displayed a window on the screen with the normal open function. This function lets the user select the size and position in advance, and permits window sizing and movement. However, Intuition offers special features for windows. Let's look at the required structure which contains all the important information for the window:

```
struct NewWindow
{
SHORT LeftEdge, TopEdge;
SHORT Width, Height;
UBYTE DetailPen, BlockPen;
ULONG IDCMPFlags;
ULONG Flags;
struct Gadget *FirstGadget;
struct Image *CheckMark;
UBYTE *Title;
struct Screen *Screen;
struct BitMap *BitMap;
SHORT MinWidth, MinHeight;
SHORT MaxWidth, MaxHeight;
USHORT Type;
}
```

The NewWindow structure

The structure definition can be found in the header file intuition/intuition.h. Now let's filter out the items that we are really going to use. The name is already remarkable: NewWindow. The same file usually contains a structure named Window. You need the NewWindow structure to define a new user window.

The first four entries represent the upper left corner and the window's dimensions in height and width. These are the same values as those used in the CLI's Open command:

```
open("CON:20/40/200/50/windowtitle", 0, 0);
```

To use the same values for the Intuition window, the values are assigned to the structure components. The following definition precedes the window description:

```
struct NewWindow NewWindow;

....

NewWindow.LeftEdge = 20;
NewWindow.TopEdge  = 40;
NewWindow.Width    = 200;
NewWindow.Height   = 50;
```

The capabilities a CLI Open command can't provide are color settings for the window.

The values in DetailPen specify the window title and the height of the title bar. The number describes the color register index. If 4 possible colors are available, they are numbered from 0 to 3. The colors cannot be changed because they're determined by the settings in Preferences. The background always occupies register 0. The BlockPen register draws the color for the window's border.

25.2.1 The Window Flags

The next entry (IDCMPFlags) will be skipped since it isn't used. The element Flags in the NewWindow structure determine certain items in the window, which are set with defines. Every define determines whether or not an Intuition function is required (e.g., WINDOWSIZING). Look at the defines used in the sample program.

There are many more useful defines included for windows.

Smart_Refresh This causes the Amiga to control and store everything concerning window changes and contents. If another window is dragged over the user window, a part of the window may be temporarily obscured. The computer automatically stores this area in a buffer and, if needed, will restore it again.

Activate This automatically activates the window once its opened. This saves the user the trouble of clicking the window to activate it.

windowsizing Permits a window with a size gadget to change dimensions. By selecting this define, the gadget appears in the lower right corner of the window. The operating system keeps watch over this gadget for any activity.

windowdrag The window can be moved.

windowdepth The window can be moved in front of or behind other windows with the front and back gadgets (the gadgets in the upper right corner). The operating system keeps watch over these gadgets for any activity.

Nocarerefresh Practically anything that occurs during program execution can selectively generate a message from the operating system. If you want a message displayed when you try resizing a window, you can generate one. For example, when editing a text using ED, the text file must be updated when the window size changes. The program gets the message that the window must be brought to the current condition. With the define above, the operating system is told that no such message is

desired. The message is not needed since SMART_REFRESH takes care of all update work when active.

Of the remaining elements of the NewWindow structure, only Title and the last five are of interest. As the name implies, the title is placed into the title bar at the top of the window. The following program demonstrates that only the address of the string is assigned:

```
NewWindow.Title = "The User window";
```

If no constants are used as in the example, the memory space must be prepared by the user and its beginning assigned to the entry Title.

The values MinWidth, MinHeight, MaxWidth, MaxHeight indicate the minimum and maximum values for the user window. The user cannot go beyond these limitations. Since the maximum height of a window is 200 or 400 pixels, these numbers are used more frequently. PAL versions of the Amiga can create a maximum height of 256 or 512 pixels, depending on the mode selected.

Finally WBENCHSCREEN is entered into the element Type so that the parameters preset by the Workbench can be used.

25.2.2 Opening a Window

After this preliminary work, the window can finally be opened. The OpenWindow function opens the window and returns a pointer. This pointer points to a window structure but shouldn't be confused with NewWindow. It is more comprehensive than NewWindow and can be examined in intuition.h.

The OpenWindow function requires the address of the NewWindow structure as the parameter. Since many compilers permit the passing of whole structures, the following expression can be used to determine the address:

```
'&NewWindow
```

If everything was processed properly, the new window will have the specifications which were entered in NewWindow.

The CloseWindow function is all you need to close the window again (e.g., when the program is finished). It has the window pointer as its only parameter. The window disappears again.

Finally the Intuition library closes to leave everything the way it was found. If you close Intuition before closing the last window, the Guru Meditation appears.

25.2.3 A Window Program

This listing describes a large amount of theory. The program opens a window which can be moved, sized and moved to the front or back. Watch upper and lowercase letters in this listing.

```
/* window_intuition.c 25.2.3  */
/* From Amiga C for Beginners */
/* by Abacus                  */

#include <exec/types.h>
#include <intuition/intuition.h>

extern struct window *OpenWindow();/* Declaration */
extern long *OpenLibrary();  /* Hello Aztec-User */
struct IntuitionBase *IntuitionBase;

#define INTUITION_REV 0

main()
{
 struct NewWindow NewWindow;
 struct Window *Window;
 long i;

 IntuitionBase = (struct IntuitionBase *)
  OpenLibrary("intuition.library", INTUITION_REV);

 if(IntuitionBase == NULL)
  exit(FALSE);

 NewWindow.LeftEdge  = 20;
 NewWindow.TopEdge   = 20;
 NewWindow.Width     = 200;
 NewWindow.Height    = 80;
 NewWindow.DetailPen = 0;
 NewWindow.BlockPen  = 2;
 NewWindow.IDCMPFlags = NULL;
 NewWindow.Flags = SMART_REFRESH | ACTIVATE |
   WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH |
   NOCAREREFRESH;
 NewWindow.FirstGadget = NULL;
 NewWindow.CheckMark = NULL;
 NewWindow.Title     = (UBYTE *)"The User window";
 NewWindow.Screen    = NULL;
 NewWindow.BitMap    = NULL;
 NewWindow.MinWidth  = 80;
 NewWindow.MinHeight = 25;
 NewWindow.MaxWidth  = 640;
 NewWindow.MaxHeight = 200; /* PAL - change to 256 */
 NewWindow.Type = WBENCHSCREEN;
```

```
 if((Window = OpenWindow(&NewWindow)) == NULL)
  exit(FALSE);
 for(i = 0; i < 800000; i++) /* Small Pause */
   ;

 CloseWindow(Window);
 CloseLibrary(IntuitionBase);
 exit(TRUE);
}
```

The program above exactly follows the indications and requirements previously stated. First the Intuition library opens. If this is not possible for some reason, a 0 is returned as an Intuition pointer. Continuing would make no sense at that point so the program will end. The window pointer returned can be null so it must be tested.

Examine the declaration of the `OpenWindow` function at the beginning of the listing. This not only preserves a good C style, but also suppresses the changing of returned values with the `cast` statement. For example, this is the case in the `OpenLibrary` because the routine returns not only the Intuition pointer, but also a variety of other pointer types. Nevertheless the function should be declared at least as a routine which returns pointers. This prevents, for example, the Aztec system crashes which result from not declaring the function. This means that for the compiler, integer values will be delivered which in Aztec are only 2 bytes long. It should be obvious that the value with 4 bytes does not arrive. Only 2 bytes will be accepted.

The `cast` statement `struct IntuitionBase*` can be used to trick the best compiler. Most of the time the resulting value isn't equal to 0 and does not cause the program to crash. During the next access to a library function with the `IntuitionBase` pointer, the system crashes. This accident cannot occur with Lattice since its integers always use 4 bytes.

During the opening of the library, the routine wants a version number which is required for proper processing. If the system has the same or later version, everything proceeds smoothly. Since this program has no special needs, a 0 is selected.

While opening the window, the address of the `NewWindow` structure is passed. The system then accepts the data in an internal area so that even this variable, `NewWindow` is no longer required. If something is done to the window, it can be done with the window structure.

If the compilation displays warning messages, don't be alarmed. Some structures need definition which are not used explicitly but appear in a structure definition as sub-elements. Maybe the functions required can be included with `include`. This may also result in additional unknown structures surfacing which also want to be defined. The only solution is to define all functions with `include`. This requires an enormous amount of memory space, and increases the compiler time

significantly. Unless the user has a RAM disk on which all `include` files are stored, it is not advisable to do this because nothing changes in the object code anyway.

After the general framework of the window program has been constructed, some experimentation is helpful. Change a few values in the `NewWindow` structure to see the effect on the window. Leave the unknown structure entries and the type element untouched.

Under Intuition, it's impossible to exhaust the topic of windows. The user who wants to know more should obtain more literature on this subject.

25.3 Screens

A screen is simply a CRT (Cathode Ray Tube) display. In most PCs and home computers, only one screen shows the screen contents. The Atari ST and Amiga screens can display several windows at a time.

The Workbench screen is already familiar to you. Any number of windows can be opened on any screen, depending on the amount of memory available. It also determines the color composition and the number of colors available to be used by the windows. The Workbench generally offers 4 different colors and works with a resolution of 640*400 pixels (640*512 pixels in PAL systems).

The user can specify these values for each program in order to construct a screen to personal taste. The number of colors depends on the number of available bit-planes. A bit-plane represents a part of memory which is used for storing graphics. More memory permits more bit-planes and therefore more colors. The Workbench uses two bit-planes for four colors. A table illustrates the connection between colors and bit-planes.

Number of bit-planes	->	Number of colors
1	->	2
2	->	4
3	->	8
4	->	16
5	->	32 (not always possible)

Not only can the number of colors be selected, but also the resolution of the screen. You can reach a maximum of 32 colors and a resolution of 640*400 pixels (640*512 pixels in PAL systems.) As in the window, two color registers can be assigned, which are responsible for the borders and the background. Since these values which are stored in the `NewScreen` structure strongly remind you of the `NewWindow` structure, let's look at the structure definition.

```
struct NewScreen
{
SHORT LeftEdge, TopEdge, Width, Height, Depth;
UBYTE DetailPen, BlockPen;
USHORT ViewModes;
USHORT Type;
struct TextAttr *Font;
UBYTE *DefaultTitle;
struct Gadgets *Gadgets;
struct bitMap *CustombitMap;
}
```

The first entries in this structure have the same names as those in NewWindow. They also have exactly the same meanings. Depth indicates the number of bit-planes (1-5), which were already discussed. DetailPen and BlockPen are the color indices. They depend on the number of available bit-planes.

The next important entries are Type (the CUSTOMSCREEN must be set here) and DefaultTitle which points to the title line of the screen. This is enough for the user program to fully define a screen.

The beauty of Intuition is that everything follows a certain pattern so that many different problems can be solved in the same manner. After understanding how to create a window, it shouldn't be a problem to create a screen on the monitor. First the NewScreen structure is stored in the manner described above. Then the screen is opened with the following function:

```
Screen = Openscreen(&NewScreen);
```

The variable Screen represents a pointer to the structure named screen. Also, the structures NewScreen and Screen must be differentiated here. NewScreen is only required once for the OpenScreen function. The data is transferred to the Screen structure (the Screen structure is much more comprehensive than NewScreen). A pointer to the new screen structure is the return value.

To open a window to this screen, the initialization of the NewWindow structure must be changed slightly. The Type in the define WBENCH-SCREEN is replaced by CUSTOMSCREEN. The entry Screen must be supplied with a screen pointer. The define CUSTOMSCREEN indicates that the window to the user screen should be opened. The window gets all the capabilities offered by the screen. Since several screens can be opened by a program, the window must be attached to a specific screen. This assignment can only be made after the screen has already been opened and the screen pointer is available.

Before the program end the screen is closed with CloseScreen to which the screen pointer is passed. The window must be closed before the screen, or serious problems will occur.

25.3.1 A Screen Program

The listing for the subject of screens has some more enhancements which will be explained now.

```
/* screen_intuition.c 25.3.1  */
/* From Amiga C for Beginners */
/* by Abacus                  */
```

```
#include <exec/types.h>
#include <intuition/intuition.h>

extern LONG OpenLibrary();
extern struct Screen *OpenScreen();
extern struct Window *OpenWindow();

struct IntuitionBase *IntuitionBase;

#define INTUITION_REV 0

struct NewScreen NewScreen =
{
   0,0,
   640, /* Width */
   200, /* Height; PAL version-change 200 to 256 */
   3,   /* 3 bitplanes = 8 colors */
   3,5, /* another color combination */
   HIRES,
   CUSTOMSCREEN,
   NULL,
   "To end the program, please click Close-Gadget!",
   NULL,
   NULL,
};

struct NewWindow NewWindow =
{
   40, 40,    /* X and Y Position */
   280, 120,  /* Width, Height */
   4, 6,      /* Colors (0 - 7) */
   CLOSEWINDOW,
   WINDOWCLOSE | SMART_REFRESH | ACTIVATE | WINDOWSIZING
     | SIZEBRIGHT | WINDOWDRAG | WINDOWDEPTH,
   NULL,
   NULL,
   "*** Hello ***",
   NULL,
   NULL,
   190, 20,
   640, 200, /* in PAL systems change the 200 to 256 */
   CUSTOMSCREEN
};


main()
{
   struct Screen *Screen;
   struct Window *Window;

   if((IntuitionBase = (struct IntuitionBase *)
     OpenLibrary("intuition.library", INTUITION_REV))
     == NULL)
     exit(FALSE);
```

```
    if ( (Screen = OpenScreen(&NewScreen) ) == NULL)
      exit(FALSE);

  NewWindow.Screen = Screen; /* Do not forget! */

  if( (Window = OpenWindow(&NewWindow) ) == NULL)
    exit(FALSE);

  /* Wait for Close-Gadget */
  Wait(1 << Window->UserPort->mp_SigBit);

  printf("\nLast window values: %d/%d/%d/%d\n\n",
    Window->LeftEdge,
    Window->TopEdge,
    Window->Width,
    Window->Height  );
  CloseWindow(Window); /* Close everything in sequence*/
  CloseScreen(Screen);
  CloseLibrary(IntuitionBase);
  exit(TRUE);
}
```

Since C programmers are usually too lazy to type, structure initialization is best performed during the definition of the variables. Another innovation is the entry of `CLOSEWINDOW` in `IDCMPFlags`. In combination with the new `WINDOWCLOSE` in `Flags`, the close gadget can be tested. The `Wait` statement tells the system to wait for the activities entered in the `IDCMPFlags`:

```
Wait(1 << Window->UserPort->mp_SigBit);
```

Clicking the close gadget is the only way out. As in the window program, all flags are set, which permit the user to change the size and position of the window. The window pointer points to the desired elements `LeftEdge`, `TopEdge`, `Width` and `Height`.

Although the example program above uses three bit-planes, the maximum accessible colors is eight. Therefore, the color registers can be from 0 to 7. Experimenting with screens takes up a lot of memory. The program above requires almost 80K of RAM.

25.4 Text/Graphic Window Display

Intuition sees little difference between processing text or graphics. Since a `printf` call doesn't work in Intuition windows, something else must be used.

25.4.1 Text

The `Text` function handles string output. A pointer to a `RastPort` structure passes the text to the window structure. The user doesn't have to know the appearance of `RastPort` or what function it performs. It is enough to pass the expression to the responsible routine:

```
Window->RPort
```

`Text` requires a character string and its length as additional parameters. The format is as follows:

```
Text(Window->RPort, string, length);
```

25.4.2 Move

The string passed by text appears at the current cursor position. The `Move` function sets this position using this syntax:

```
Move(Window->RPort, xpos, ypos);
```

Before each call of the `Text` function, `Move` should position the cursor. A small routine for this task follows:

```
text(w_ptr, s, x, y)
struct Window *w_ptr;
char *s;
int x, y;
{
     Move(w_ptr->RPort, x, y);
    Text(w_ptr->RPort, s, strlen(s));
}
```

To keep the function generic, the pointer to the window in which the text should appear is passed. Because of this you can service several windows with the same function. A call appears as follows:

```
text(Window, "Attention!", 20, 40);
```

The text appears at position (20/40), if the window will allow it (the if is important). You can write as much text as you wish in the window. Intuition ensures that no window or screen is overwritten. If the text can't be displayed completely in the window, the writing stops at the window's right border. The user can be assured that nothing is accidentally drawn in other windows.

25.4.3 Draw

The `Draw` function draws lines. The parameters of the routine are:

```
Draw(Window->RPort, x, y);
```

Something's missing here—you need two points to draw a line. With `Draw`, the straight line is drawn between the current position and the (x/y) point. The `Draw` function belongs to the `graphics.library` instead of `intuition.library`. First this library must be opened, then it returns a special pointer.

For drawing, the mouse coordinates are normally needed. They are found in `mouseX` and `mouseY` which are two elements of the window structure. This includes everything needed to write into a window.

In the following program `argv` and `argc` reappear. These other values can be used from the `CLI` rather than the preset values. The call has the following format:

```
prg X-RES Y-RES BITPLANES
```

for example:

```
draw 640 200 3
```

It is interesting here that the resolution of a screen can be larger than the maximum resolution of the display monitor. 640*200 pixels (640*256 in PAL systems) can be represented, but if 800 dots are selected in the horizontal axis, the window can be shifted beyond the right screen border. The same is true of the vertical axis.

25.4.4 Small Drawing Program

```
/* draw.c 25.4.3  */
/* From Amiga C for Beginners */
/* by Abacus                  */

#include <exec/types.h>
#include <intuition/intuition.h>

extern LONG OpenLibrary();
extern struct Screen *OpenScreen();
extern struct Window *OpenWindow();

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

#define INTUITION_REV 0
#define GRAPHICS_REV 0

struct TextAttr Font =
{
   "topaz.font",
   TOPAZ_SIXTY,
   FS_NORMAL,
   FPF_ROMFONT,
};

UBYTE screentitle[81];


struct NewScreen NewScreen =
{
   0,0,
   640, /* Width */
   200, /* Height; PAL version-may change 200 to 256 */
   2,   /* 3 bitplanes = 8 colors */
   2,3, /* another color combination */
   HIRES,
   CUSTOMSCREEN,
   &Font,
   screentitle,
   NULL,
   NULL,
};

struct NewWindow NewWindow =
{
   20, 20,   /* X and Y Position */
   400, 180,  /* Width, Height */
   0,1,     /* Colors (0 - 7) */
   CLOSEWINDOW,
   WINDOWCLOSE | SMART_REFRESH | ACTIVATE | WINDOWSIZING |
```

```
        SIZEBRIGHT | WINDOWDRAG | WINDOWDEPTH,
    NULL,
    NULL,
    "* My window *",
    NULL,
    NULL,
    190, 20,
    640, 200, /* in PAL systems may change 200 to 256 */
    CUSTOMSCREEN
};


main(argc,argv)
int argc;
char *argv[];
{
    struct Screen *Screen;
    struct Window *Window;
    register char s[81];
    int color = 4;
    register int x, y, xalt, yalt;

    if((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library", INTUITION_REV))
      == NULL)
      exit(FALSE);

    if ((GfxBase = (struct GfxBase *)
       OpenLibrary("graphics.library",GRAPHICS_REV))
       == NULL)
       exit(FALSE);

    if (argc != 4)
       {
          printf("Error in arguments\n");
          printf("X-Res Y-Res Bitplanes\n");
       }
    else
       {
          NewScreen.Width    =  atoi(argv[1]);
          NewScreen.Height   =  atoi(argv[2]);
          NewScreen.Depth    =  atoi(argv[3]);
          if(NewScreen.Depth > 4 || NewScreen.Depth < 1)
             NewScreen.Depth = 2;
          color = 1 << NewScreen.Depth;
          NewScreen.DetailPen  =  color - 1;
          NewScreen.BlockPen   =  color - 2;
       }



    sprintf(screentitle,"This screen has %d colors",
            color);

     if ( (Screen = OpenScreen(&NewScreen) ) == NULL)
       exit(FALSE);
```

```
    NewWindow.Screen = Screen; /* Do not forget! */

    if (argc == 4)
       {
          NewWindow.Width      =  Screen->Width/2;
          NewWindow.Height     =  Screen->Height/3;
          NewWindow.MinWidth   =  Screen->Width/3;
          NewWindow.MinHeight  =  Screen->Height/5;
          NewWindow.MaxWidth   =  Screen->Width;
          NewWindow.MaxHeight  =  Screen->Height;
       }


    if( (Window = OpenWindow(&NewWindow) ) == NULL)
      exit(FALSE);

   text(Window,"Hello there!",20,20);

    /* Initialize with start values */
    Move(Window->RPort, xalt = Window->MouseX,
          yalt = Window->MouseY);

    /* Drawing starts here until upper or */
    /* left  border is reached */
    while( (x = Window->MouseX) > 0 &&
             (y = Window->MouseY) > 0)
       {
          sprintf(s,"X = %3d, Y =%3d", x , y);
          text(Window, s, 150, 7);
          Move(Window->RPort, xalt, yalt);
          Draw(Window->RPort, xalt = x, yalt = y);
       }

   text(Window, "Please click the close gadget", 20, 20);


    /* Wait for Close-Gadget */
    Wait(1 << Window->UserPort->mp_SigBit);

    CloseWindow(Window);
      /* Close everything in sequence*/
    CloseScreen(Screen);
    CloseLibrary(GfxBase);
    CloseLibrary(IntuitionBase);
    exit(TRUE);
}

text(w_ptr, s, x, y)
struct Window *w_ptr;
char *s;
int x, y;
{
    Move(w_ptr->RPort, x, y);
    Text(w_ptr->RPort, s, strlen(s));
}
```

Notice the new structure `TextAttr` with the variable `Font`. Every `NewScreen` structure has, among other things, a component named `Font`. Here the character set to be used can be stored (address of the structure). Of course this is again a pointer to another structure, `TextAttr`.

The structure for the definition of a character set is very simple. First there is the name of the font, then the height of the character and the manner of presentation. Finally a flag marks the location of the character set. We used the character set built into ROM. This font is used in the 60 character screen setting. It's somewhat larger than the 80 character version. Eighty characters can be displayed if you use `TOPAZ_EIGHTY` instead of `TOPAZ_SIXTY`.

All entries of `NewWindow` and `NewScreen` are already initialized, but if the user wants to use other values for resolution and bit-planes, they can be accepted. The maximum and minimum size of the windows must also be adjusted. Since the color indices of the window are always ready for use with 0 and 1, no conversion has to be performed. The screen, on the other hand, gets the last two color registers which of course depend on the number of colors. The program permits a maximum of 16 colors, which equals four bit-planes, in the high resolution mode. The title of the screen displays this information. For this reason an additional variable must be used because the structure has made no provisions for storing the title.

As soon as the window is opened, the text appears. The mouse can be used to draw something. As long as the mouse pointer doesn't move beyond the upper or left border of the window, a line will be drawn. The mouse position is always indicated in the title line. This is not absolute, but relative to the upper left corner of the window. For this reason a negative value is possible when the mouse is pushed beyond the left or upper border. The program uses this as an end criterion.

Because of the output of the mouse coordinates, the current character position in `xalt` and `yalt` must be stored in intermediate storage. Before the drawing of the line, the values are restored again with the `Move` function. We have said enough about this program.

However, we have some additional suggestions. Up to now only the define `HIRES` was used for user defined screens in `ViewModes`, which achieves a horizontal resolution of up to 640 pixels. The Amiga can also produce a lower resolution which fills a complete display line with 320 pixels. With the same number of colors, only half the memory space is required. The possibility of working with five bit-planes and, therefore, 32 different colors is created. The only thing required for this is to replace `HIRES` with `NULL`. Try this on the first screen program. Besides the change mentioned above, only the width of the screen must be set to the 320 pixels. If five bit-planes were requested, 32 colors are available in color registers 0-31.

25.4.5 Low Resolution and Interlace Modes

The interlace mode can be switched on in `ViewModes`. The display gets a vertical resolution of 400 (PAL systems—512) instead of 200 (PAL—256) pixels. Not bad, but here comes the big "but!" This process can only be realized by lowering the refresh frequency of the monitor from 50 to 25 Hz. A flickering image occurs on the Amiga monitor. This mode was only intended for those monitors which had a long screen refresh rate and thereby don't create flickering. Examine the display and form your own judgment. The define `LACE` in the `ViewModes` enters interlace mode. Some examples:

Low resolution (320 pixels):

```
NewScreen.ViewModes = NULL;
```

High resolution (640 pixels) and interlace (400 pixels):

```
NewScreen.ViewMode = HIRES | LACE;
```

Low resolution (320 pixels) and interlace (400 pixels):

```
NewScreen.ViewModes = LACE;
```

With 32 colors, a function which permits changing the color of the pen would be useful. This input changes the current color in the color register:

```
SetAPen(Window->RPort, color);
```

To see this routine in action, a new program must be written. Add the following lines to the drawing program. First, at the beginning of the `main` function, add the definition of a variable named `colour`:

```
register int x, y, xalt, yalt, colour = 1;
```

At the end of the `while` loop add two additional lines. Here is the complete loop :

```
while( (x = Window->mouseX) > 0 &&
        (y = Window->mouseY) > 0)
{
sprintf(s, "X =%3d, Y = %3d, x, y);
text(Window, s, 150, 7);
Move(Window->RPort, xalt, yalt);
Draw(Window->RPort, xalt = x, yalt = y);

SetAPen(Window->RPort, colour++); /* New! */
if(colour == color) colour = 1;  /* New! */
}
```

This program is on the optional disk as `draw16.c`. In the beginning the variable `colour` is set at 1 to draw with the first color index (0 is the background color). After each small line fragment, the pen color changes until the last index (number of colors-1) has been reached. Then it starts over. This produces all the colors for the first run, including the registers not normally used. To make it even more colorful than the 16 colors in the `HIRES` mode, change the safety test:

old:

```
if(NewScreen.Depth > 4 || NewScreen.Depth < 1)
     NewScreen.Depth = 2;
```

to

new:

```
if(NewScreen.Depth > 5 || NewScreen.Depth < 1)
     NewScreen.Depth = 2;
```

and change the `HIRES` to `NULL`. Consider the reduced X resolution of 320 pixels instead of 640 pixels and change the program accordingly. This permits drawing with 32 colors in one window. This program is on the optional disk as `draw32.c`. An example call of the program for 32 colors:

```
draw32 320 200 5
```

Since many colors and high resolution require a large amount of memory, here are some examples of memory usage by windows and screens.

Low resolution, interlace mode, 32 colors:

```
320 (pixels) * 400 (pixels) * 5 (bitplanes) / 8 (bits per
byte) = 80,000 bytes = 78K (PAL 320*512*5/8 = 102,400)
```

High resolution, interlace mode, 8 colors:

```
640 (pixels) * 400 (pixels) * 3 (bitplanes) / 8 (bits per
byte) = 96,000 bytes = 93K (PAL 640*512*3/8 = 122,880)
```

Low resolution, 2 colors

```
320 (pixels) * 200 (pixels) * 1 (bitplanes) / 8 (bits per
byte) = 8,000 bytes = 7.8K (PAL 320*256*1/8 = 10,240)
```

25.4.6 Pixel Processing

Other graphic commands besides `Draw` are:

```
ReadPixel(Window->RPort, x, y);
WritePixel(Window->RPort, x, y);
```

ReadPixel ReadPixel tests whether the indicated position of a pixel was set and returns the color value of the pixel. If no pixel was visible, because the pixel has the same color as the background, ReadPixel returns a 0 to the register number. If the pixel is outside the window whose port is passed, the result is -1.

WritePixel WritePixel sets a single pixel at the position indicated. The color used depends on the current color as in all other routines and is determined by SetAPen.

The following is a program which changes the title line of the window used in a sine format.

```
/* pixel.c 25.4.6              */
/* From Amiga C for Beginners */
/* by Abacus                   */

#include <exec/types.h>
#include <intuition/intuition.h>

extern struct Window *OpenWindow(); /* Declaration */
extern long *OpenLibrary();  /* Hello Aztec-User! */
extern double sin();

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

#define INTUITION_REV 0
#define GRAPHICS_REV 0

/* Number of colors of Workbench */
#define WB_COLORS 4

struct NewWindow NewWindow =
{
   10, 50,    /* X and Y Position */
   360, 120, /* Width, Height */
   3, 2,      /* Color Indexes */
   NULL,
   SMART_REFRESH | ACTIVATE | WINDOWDRAG | WINDOWDEPTH,
   NULL,
   NULL,
   "This Line is changed!",
   NULL,
   NULL,
   0, 0,
   640, 200, /* PAL users - change 200 to 256 */
   WBENCHSCREEN
};


main()
{
   struct Window *Window;
   register struct RastPort *r;
```

```
    register int i, j, top, yoffset;
    int i_to, j_to, color, colors[512];
    double factor;

    if((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library", INTUITION_REV))
       == NULL)
      exit(FALSE);

    if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library", GRAPHICS_REV) )
        == NULL)
       exit(FALSE);

    if((Window = OpenWindow(&NewWindow)) == NULL)
      exit (FALSE);

    r = Window->RPort;
    top = Window->Height / 4;
    factor = 2 * 3.1415926 / Window->Width * 1.5;
      /* 1.5 Sine Waves */

    for(i = 2, i_to = Window->Width - 2; i < i_to; i++)
      {
         for(j = 0; j < top; j++) /* A vertical line */
           {          /* Transfer to Array */
                color = ReadPixel(r, i, j);
                if(++color == WB_COLORS)
                   colors[j] = 0;
                else
                   colors[j] = color;
                   /* increase color index by one */
          }
         for(j = 0,
            yoffset = top + top * sin(factor * i) + 16;
            j < top; j++)
            if(colors[j])  /* If Pixel should be set */
              {
                 SetAPen(r, colors[j]);
                 WritePixel(r, i, j + yoffset);
              }
      }
    Delay(1500);     /* Wait 1500 Ticks = 30 seconds */
    CloseWindow(Window);
    CloseLibrary(GfxBase);
    CloseLibrary(IntuitionBase);
    exit(TRUE);
}
```

Lattice

The library for mathematical functions and floating point numbers must be linked with the standard library. Example:

```
lc -Lm  math2
```

Aztec

If you work with the Aztec compiler, the library for mathematical functions and floating point numbers must be linked with the standard library `c.lib`. Example:

```
cc +L math2.c
ln math2.o -lm -lc
```

How the program works:

The program runs under the Workbench screen and uses its colors. Two bit-planes (four colors) are the default. If the user constructed a Workbench which deviates from this, the `define WB_COLORS` must be adjusted accordingly. The window which was defined in the `NewWindow` structure contains only front and back gadgets. The size cannot be changed.

In a large loop which processes the complete width of the window, all pixels belonging to one X position are gathered in one array. Before the value which is returned from the `ReadPixel` is stored, a color transformation is made. Every pixel gets the color from the following register. The area of the display which is transmitted, is the upper quarter of the window. To prevent mix-ups between the information to be read and written, a complete column is first saved into the array `colors`. Then the new position of the pixels is calculated using the `sin` (sine) function. The dots are written in the new color with `WritePixel` at the new position. The pixels which are the same color as the background are taken out. This is tested first to increase the speed. The program will display the top line as a sine wave.

After complete transformation, the `Delay` function slightly delays the end of the program to give the user the opportunity to view the window again. The parameter provides the waiting time in 1/50 second ticks. To achieve a delay of 30 seconds 1,500 ticks must be stored.

25.5 DOS

Besides Intuition, DOS (the Disk Operating System) is usually required when programming. Many programs wouldn't work without the help of AmigaDOS. The DOS routines handle file deletion and renaming, as well as directory creation and directory display.

DOS is also stored in a library file (`dos.library`). Unlike the other libraries, `dos.library` is always open to the user. It doesn't have to be opened or closed by the user. It is as simple to use as the functions from the standard library. For example, deleting a file:

```
result = DeleteFile(filename);
```

The return value is an integer number which indicates an error if zero and the correct processing with a number unequal to zero. Complete directories can be deleted with this function. No files may be contained in the directory. They must be erased in advance. The `Rename` function is just as easy to use for renaming files:

```
result = Rename(old_name, new_name);
```

The result is also a value unequal to zero. The other parameters `old_name` and `new_name` are just like the `filename` strings which contain a valid filename.

25.6 SetComment

One routine permits the attachment of a comment to a file. This comment is completely independent of the content and size of the file and is stored in the same place as the filename and its parameters. The DOS command FILENOTE in the C: directory can attach a comment text to an existing file. This can also be done with the function:

```
result = SetComment(filename, comment);
```

A short program shows this routine in action:

```
/* makecom.c 25.6              */
/* From Amiga C for Beginners */
/* by Abacus                  */

#include <libraries/dos.h>

main(argc, argv)
int argc;
char *argv[];
{
   if(argc == 3)
     {
      if(!SetComment(argv[1], argv[2]))
      printf("Error %d\n", IoErr());
     }
   else
     printf("Format: MAKECOM FILE COMMENT\n");

   exit(TRUE);
}
```

The filename and the comment passes to the program through the command line. As in all data transfers, the use of spaces in the actual comment is not permitted. Here is a sample call:

```
makecom makecom.c This-is-a-comment-use-the-list-command-to-display-it
```

25.7 Read Directory

Many programs that handle files should have a routine for reading disk directories. To make this possible in a program, various functions are required.

First there is the `Lock` function. `Lock` locks the specified directory for access. Only after the `Lock` can other functions operate on the directory. The name of the directory and the access mode are passed to `Lock`. Finally an integer value for `read` is passed with the define `ACCESS_READ`. The returned key permits the processing of this one directory, similar to a handle for file accesses or a window pointer under Intuition. If this key is equal to zero, an error has occurred.

Two functions are required for reading a directory. One is `Examine`, the other `ExNext`. First `Examine` must be called to obtain the first entry of the directory. Then a call for `ExNext` must follow for each additional file or directory. Both routines require both `Lock` and a pointer to the `FileInfoBlock` structure. This structure contains all important file data.

```
result = Examine(lock, &fileinfo);
```

And here is the structure definition:

```
struct FileInfoBlock {
LONG fib_DiskKey;
LONG fib_DirEntryType;
char fib_FileName[108];
LONG fib_Protection;
LONG fib_EntryType;
LONG fib_Size;
LONG fib_NumBlocks;
struct DateStamp fib_Date;
char fib_Comment[116];
}
```

The `fib_DirEntryType` function indicates whether the data currently read is a normal file (<0) or a directory (>0).

The `fib_FileName` function contains the name which can be up to 30 characters long, even though it was generously defined here as 108 characters.

The `fin_Protection` function contains flags which indicate whether the file can be read, written, executed or erased. The variable is defined as `LONG` (32 bits), but only the lower 4 bits are required.

The priorities for these files are as follows:

```
      R          W          E          D
      8          4          2          1
R = Read   W = Write  E = Execute  D = Delete
```

For every protective action one of the bits above must be set. For example if a file or directory can only be read or erased, the flags W and E must be set:

```
RWED
0110 (bits) = 4 + 2 = 6
```

This variable must contain the value 6. It is important to set the flag whose function is forbidden. To make changes please use the PROTECT command from the CLI. The flags which are passed with this program are changed in such a manner that the functions can be performed. This is exactly the opposite of their use in the user program. To protect a file from erasing, the following line is required:

```
PROTECT ED RWE
```

The `fib_Size` function defines the file size in bytes.

The `fib_NumBlocks` function contains the number of occupied blocks on the disk.

The `fib_Date` function contains the date when the file was last written.

The `fib_Comment` function contains the comments for the file.

You now have all the information needed to construct the final program. For the sake of simplicity we use parameter passing with the command line in this version. This parameter indicates the directory which should be read.

```
/* read_dir.c 25.7              */
/* From Amiga C for Beginners */
/* by Abacus                    */

#include <libraries/dos.h>

struct FileInfoBlock fi;

main(argc, argv)
int argc;
char *argv[];
{
   long lock;
   int error;
   char filepath[100];

   if(argc == 2)   /* parameter present ? */
```

```
        strcpy(filepath, argv[1]);
    else
        strcpy(filepath, "sys:");

    lock = Lock(filepath, ACCESS_READ);
    printf("Lock value %d\n", lock);
    if(!lock)
        {
            printf("No Lock! ERROR!\n");
            exit(FALSE);
        }

    if(Examine(lock, &fi))  /* First call successful? */
        do
          output(); /* Return value not of interest now */
        while(ExNext(lock, &fi)); /* until error occurs */

    error = IoErr();    /* What Error? */
    if(error != ERROR_NO_MORE_ENTRIES) /* "real" Error! */
        printf("Error %d occurred!\n", error);

    exit(TRUE);
}

output()
{
    if(!*fi.fib_FileName) /* strlen = 0 */
      {
        printf("Empty!\n");
         /* for example Root-directory of RAM Disk */
        return(0); /* That's directory without name */
      }
    if(fi.fib_DirEntryType > 0)
      printf("Directory name");
    else
      printf("Filename      ");

  printf(": >%20s< RWXD %lx bytes: %-6ld Blocks %-4ld\n",
          fi.fib_FileName, fi.fib_Protection,
          fi.fib_Size, fi.fib_NumBlocks);

    if(*fi.fib_Comment) /* If comment present, output! */
        printf("Comment: >%s<\n", fi.fib_Comment);
    return(fi.fib_DirEntryType > 0); /*Return File Type */
}
```

The protect flags are not decoded separately in this program (it wouldn't be a problem to do that), but are presented as a hexadecimal number (the "`%lx`" format instruction). You can enhance this program if you wish.

The flags can be displayed in RWED format using the `LIST` command from the `CLI`. With `PROTECT` any flag can be set and with `LIST` the result can be observed. This information can be compared with the results from the user program.

25.8 Conclusion

You now have the general knowledge needed to write simple programs in C language. As you take time to develop your own programs, functions and libraries, keep this book nearby for reference. Since it's difficult to memorize everything about a language, this book will help you with the complex concepts of C language.

You may be wondering why we didn't spend more time with some aspects of the Amiga. We admit that we didn't include as much about the operating system and Intuition as we would have liked. However, these are difficult concepts for a beginner to understand, and we felt it best to just give the reader a few general examples controlling these areas in C. Since C is a transportable language, you may prefer to write transportable source codes.

If you want to continue your education in C, we recommend the Abacus book *Amiga C for Advanced Programmers.* This book covers subjects that interest professional Amiga programmers: Combining assembly language and source codes; debugging (finding errors); jump tables; and more. In addition, *Amiga C for Advanced Programmers* details Intuition programming in C (menus, requesters, etc.).

We wish you luck in your future as a C programmer.

Dirk Schaun

Appendices

A. Functions

Filename: strlen.c

```
/*******************************************************/
/* Name:          strlen                               */
/* Parameter:     s (String)                           */
/* Return value: Length (int)                          */
/* Function:    Number of characters in "s"            */
/* Other:     -                                        */
/*******************************************************/

strlen(s)
char s[];
{
  register int i = 0;
  while(s[i])
    i++;
  return(i);
}
```

Filename: strcpy.c

```
/*******************************************************/
/* Name:          strcpy                               */
/* Parameter:     s (String), t (String)               */
/* Return value: -                                     */
/* Function:      Copies "s" to "t"                    */
/* Other:        -                                     */
/*******************************************************/

strcpy(t,s)
register char *t,*s;
{
  while(*t++ = *s++)
    ;
}
```

Filename: strcat.c

```
/*******************************************************/
/* Name:          strcat                               */
/* Parameter:     s (String), t (String)               */
/* Return value:  -                                    */
/* Function:      attach "t" to "s"                    */
/* Other:         -                                    */
/*******************************************************/

strcat(s,t)
```

```
register char *s,*t;
{
  while(*s)
    s++;
  while(*s++ = *t++);
}
```

Filename: `letter.c`

```
/*******************************************************/
/* Name:          letter                               */
/* Parameter:     z (char)                             */
/* Return value: It was a letter (1), else (0)         */
/* Function: Determines if it was a letter or not      */
/* Other:     -                                        */
/*******************************************************/
#define FALSE 0
#define TRUE  1

letter(z)
register char z;
{
  if ((z >= 'a' && z <= 'z') || (z >= 'A' && z <= 'Z'))
return(TRUE);
  return(FALSE);
}
```

Filename: `c_comp.c`

```
/*******************************************************/
/* Name:          c_comp                               */
/* Parameter:     c1 (char), c2 (char)                 */
/* Return value:  1 (TRUE), 0 (FALSE)                  */
/* Function:      Compares two characters              */
/* Other:         requires two characters ()           */
/*******************************************************/
#define FALSE 0
#define TRUE  1
extern int grklflag;

c_comp(c1,c2)
register char c1,c2;
{
  if (c1 == c2) return(TRUE);
  if( grklflag && letter(c1) && letter(c2))
    if((c1 + 'a' - 'A' == c2)||(c2 + 'a' - 'A' == c1))
return(TRUE);
  return(FALSE);
}
```

Filename: strcmp.c

```
/*****************************************************/
/* Name:          strcmp                             */
/* Parameter:     s (String), t (String)             */
/* Return value: identical 0 not identical 1         */
/* Function:      Compares "s" and "t"               */
/* Other:         -                                  */
/*****************************************************/

strcmp(s,t)
register char *s, *t;
{
  register int identical;
  while(identical = c_comp(*s, *t++) )
    if(!*s++)
      return(0);
  return(!identical);
}
```

Filename: strchar.c

```
/*****************************************************/
/* Name:          strchar                            */
/* Parameter:     s (String), c (char)               */
/* Return value: Position (int), or -1               */
/* Function:      Determines Pos of the Char "c" in "s"*/
/* Other:         -                                  */
/*****************************************************/

strchar(s,c)
register char s[];
register char c;
{
  register int i = 0;
  while(!c_comp(s[i],c) && s[i])
    i++;
  if(s[i]) return(i);
  return(-1);
}
```

Filename: strchbac.c

```
/*****************************************************/
/* Name:          strchback                          */
/* Parameter:     s (String), c (char)               */
/* Return value: Index (int)                         */
/* Function: Searches for Pos of Char "c" in "s"     */
/* Other: requires c_comp(), strlen()                */
/*****************************************************/

strchback(s,c)
register char s[];
```

```
register char c;
{
  register int i = strlen(s);
  while((i >= 0) && !c_comp(s[i],c) )
    i--;
  return(i);  /* Error = -1 */
}
```

Filename: `ilatoila.c`

```
/*******************************************************/
/* Name:          ltoa                                 */
/* Parameter:     n (long), s (String)                 */
/* Return value:  -                                    */
/* Function:      Converts long value to char  string  */
/* Other:         requires reverse()                   */
/*******************************************************/
#define TRUE 1
#define EOS '\0'

ltoa(n, s)
register char s[];
register long n;
{
  register int i = 0;
  register int forechar = 0;
  if (n < 0)
    {
      forechar = TRUE;
      n = -n;
    }
  do
    {
      s[i++] = n % 10 + '0';
    } while((n /= 10) > 0);
  if(forechar )
    s[i++] = '-';
  s[i] = EOS;
  reverse(s);
}


/*******************************************************/
/* Name:          itoa                                 */
/* Parameter:     n (int), s (String)                  */
/* Return value:  -                                    */
/* Function:    Converts integer number to char string */
/* Other:         requires ltoa()                      */
/*******************************************************/

itoa(n, s)
register int n;
register char s[];
{
  ltoa((long)(n),s);
```

```
}


/*******************************************************/
/* Name:          atol                                 */
/* Parameter:     s (String)                           */
/* Return value:  n (long)                             */
/* Function:      Converts char string into long value */
/* Other:         -                                    */
/*******************************************************/

long atol(s)
register char *s;
{
  register long val;
  register int sign = 1;
  while(*s == ' ')
    s++;
  if (*s == '+' || *s == '-')
    sign = (*s++ == '+') ? 1 : -1;
  for(val = 0; *s >= '0' && *s <= '9'; ++s)
    val = 10 * val + *s - '0';
  return(sign * val);
}


/*******************************************************/
/* Name:          atoi                                 */
/* Parameter:     s (String)                           */
/* Return value:  Integer number                       */
/* Function:      Converts Char string into Integer    */
/* Other:         -                                    */
/*******************************************************/

atoi(s)
register char *s;
{
  long atol();
  return(atol(s));
}
```

B. The History of C

C originated from BCPL (Basic Cambridge Programming Language). The B language came from BCPL and C came from the B language. C was developed in the mid-seventies by Dennis M. Ritchie, who, at that time, was working for Bell Laboratories.

C was originally intended for developing an operating system which, among other things, would be capable of multi-user and multitasking execution, namely UNIX. This explains why C programs are so fast. Multitasking procedures require a very fast operating system which up to then could only be written in assembly language. Dennis Ritchie developed the C language to circumvent the error prone and unclear assembly language programming. The result, the UNIX operating system, consists of about 13,000 lines of which only a minimum of about 800 lines were written in assembly language. The rest of the operating system is in C.

C became popular with the introduction of the Amiga and the Atari ST, whose operating systems were written in C. The Amiga's Intuition user interface was written almost completely in C. Professional programmers and software houses prefer using C to develop new programming projects. C has another advantage: it is portable. This means that C programs can, theoretically, be transferred to other computers and compiled there without changes.

The reason for this is that C has a small number of commands available to all compilers. Parts which are computer specific, such as input and output, don't belong to the actual C language. These routines are delivered with the language in libraries adapted to the peculiarities of the particular computer. The C programmer doesn't have to be concerned about this. He knows that the `getchar` function gets a character from the key-board, regardless of whether the program is executing on a C64, an IBM PC, an Amiga or an Atari ST. This portability means less programming for the developers—just transfer the program over to another computer, make the changes needed for the new computer and recompile it.

How a C compiler works

Every C compiler has been split into various program portions which, depending on the manufacturer, are available either in a program module or in several smaller programs.

The first part of a C compiler is the preprocessor; this only replaces one text portion with another according to the user's commands. The result of this effort is a file containing pure text which can be processed with the editor. This result passes to the scanner which searches for command words specific to C. It recognizes these words and stores them in

abbreviated form. In this format the command is stored as a token (code) instead of as individual letters. Tokenizing takes up less memory and accelerates the translation.

Parser

After completing this test run, the parser appears. It tests the source code commands for correct syntax, and differentiates between correct and incorrect combinations of C commands. The parser knows all the rules about C syntax. Just as in everyday conversation, stringing words together isn't enough. The parser ensures that the expression is correct.

As the last part of the actual C compiler, the code generator converts the text processed by the parser into machine language commands. Some C compilers first translate the machine language commands into assembly language so the programmer can streamline the generated code. This is really unnecessary since the C compilers on the market already produce very efficient machine language code. After completing this run, the compiler saves the object file to disk with the extension of .o.

The final process is linking the object files with the required libraries to produce an executable program. The linker is used for this purpose.

C. The Lattice C Compiler

C compiler manufacturers are constantly updating and improving their compilers. For instruction on how to install the compiler, see the documentation and any README files that came with the compiler.

The C compiler

The C compiler is called with `lc` and the source filename. As a minimum the following is required:

```
lc hello
```

With the `-L` option, the linker can be loaded immediately after the call of the C compiler. The linker (here `BLINK`) can also be called independently. This is a standard call to compile and link a source code named `math2.c` into a program named `math2`.

When the library for mathematical functions and floating point numbers must be linked with the standard library, the following can be used:

```
lc -Lm  math2
```

The linker

The program `BLINK` provides a powerful linker for the programmer. Here are the most important options available in this linker:

After the name `BLINK` all files which are linked together appear after the `FROM` argument (or `ROOT`, or even nothing) . The name of the program to be executed follows the `TO` argument. Library files to be searched are listed after the `LIBRARY` argument. Some sample calls:

```
BLINK FROM a,b,c TO program

BLINK a+b+c TO program LIBRARY folder/d

BLINK ROOT a,b,c TO folder/prg LIBRARY
system/lib,obj/special
```

Using the parameter `WITH` all options can be stored in a file just as in a `MAKE` file. A linker call is then:

```
BLINK WITH file
```

In this `file` the options mentioned above contain a parameter in each new line. Example:

```
ROOT a,b,c
TO folder/prg
LIBRARY system/lib,obj/special
```

D. The Aztec C Compiler

C compiler manufacturers are constantly updating and improving their compilers. See the documentation that came with the compiler for instruction on how to install the compiler. Be sure to read any README files.

The compiler

The compiler with the name CC can be found in directory C:. The call is very simple:

```
cc file.c
```

The source file `file.c` is compiled and translated into assembly code. This code can be optimized by a machine language programmer. This file has the name `ctmpAXX.XXX`, where `X` is a number which differs from one call to the next. It is best to look at the current directory because this name is needed immediately.

Also Aztec uses symbolic names for devices which are listed as follows:

```
CLIB
INCLUDE
CCTEMP
```

CCTEMP determines where temporary files created by the C compiler will be stored. `CLIB` indicates the path to the libraries, while INCLUDE is the path for the header (`.h`) files. Examples are:

```
assign CLIB: df0:lib/
assign INCLUDE: df0:include/
assign CCTEMP: ram:
```

Various options can be placed in front of the name of the file to be compiled. Here are the most important:

`-Ipath:`
: With `-I` a pathname can be provided in which the Include files are assumed to be. The search for these files is made only in this sub-directory. The option is comparable with the assignment INCLUDE (see above).

Note:

The pathname immediately follows the `I` without any additional spaces. For example:

```
cc -Iram:includes/privat/
```

+C creates longer code, since jumps within the program code are equipped with 32-bit commands instead of the 16-bit commands which could have been used.

+D causes data to be stored in 32-bit format. This slows down data access and increases memory requirements, but makes data segments of any desired size possible (theoretically). During a "normal" data access with 16-bit addressing, the user is limited to a maximum of 64K of data.

+L Variables and constants of `int` type are stored in 32-bit format. Because of this the programs become (partially) Lattice compatible since this compiler always uses 32 bits. Without this option 16 bits are sufficient for `int` numbers.

-D defines a constant. It corresponds to the `#define` statement, but is assigned during the call of the compiler. No space can follow this option character. Example:

```
cc -DTESTRUN=1 file.c
```

The assignment corresponds to the `define`.

```
#define TESTRUN 1
```

-S suppresses "warnings." The warnings are only messages and the compiled programs are usually capable of being run. To identify the real errors, this option can be used to display only the error messages on the monitor.

+p causes the compiler to create Lattice compatible code. All data, jumps and `int` numbers are automatically created in the 32-bit version.

The assembler After the C compiler comes the assembler which is named `AS`. It is also stored in the C: directory. The call is similar to the C compiler:

```
as file.o
```

With the option `-O` a new name can be given to the new file. Example:

```
as -O program.o ctmpxyz.123
```

Additional options are only of interest to the assembler specialist. Since this book deals with C programming, they will not be considered further. It is only an intermediate step.

The linker For Aztec the linker provided is called `ln` and can be found in the C: sub-directory. The files to be linked together are placed one behind the other. Whether they are libraries or modules doesn't matter in principle, but libraries should be placed at the end of the list.

```
ln file.o c.lib
```

The standard file `c.lib` is also like all other libraries in the `lib:` directory. Several modules can be linked together:

```
ln -o result modul1.o modul2.o modul3.o c.lib
```

The name assignment for the resulting program is performed with `-o`. The linker can be informed about libraries with `-L`, but the extension of `.lib` is then omitted. Example:

```
ln file.o -Lc -Lm
```

Two additional options are `+C` and `+F` which permit selection of special storage areas. A code letter follows the option which has the following significance:

c Program

d Initialized data

b Data which was not initialized

The `+C` stands for chip memory, `+F` for fast memory. These two groups of RAM areas are especially important for graphic programming since certain data must always be stored in the chip-memory. With this option the following can be requested:

```
ln +Cdb +Fc file.o -Lc
```

This causes the storage of data in chip-memory and the storage of the program in the normal fast memory. Without a special statement, all information would have been stored in the fast memory area.

Here is a MAKE file which is tailored to Aztec C.

```
.key file

echo " Compiling  <file$t1>.c "
cc -t <file$t1>.c
echo " Assembling  <file$t1>.asm  "
as <file$t1>.asm
echo " Linking  <file$t1>.asm to <file$t1>  "
ln <file$t1>.o -lm -lc
; -lm -lc is called :link clib and mathlib in addition
echo " Everything clear !"
```

The file is named `az-make` on the optional program disk. Here is an example call to compile, assemble and link a source file named `array.c`:

```
execute az-make array
```

E. Reserved C Words

Commands which are presented here, but were not described in the book, either have no function in the current C compilers, or are reserved for future versions.

`auto`	`enum`	`short`
`break`	`extern`	`sizeof`
`case`	`float`	`static`
`char`	`for`	`struct`
`continue`	`goto`	`switch`
`default`	`if`	`typedef`
`do`	`int`	`union`
`double`	`long`	`unsigned`
`else`	`register`	`void`
`entry`	`return`	`while`

F. Operator Precedence

Precedence	Operator	Description	Evaluation
1	()	Function	left to right
	[]	Array	left to right
	.	Structure declaration	left to right
	->	Structure declaration (pointer)	left to right
2	cast	Forced type conversion	right to left
	*	Content of	right to left
	&	Address of	right to left
	-	Negative sign	right to left
	!	Logical NOT	right to left
	~	Bitwise complement	right to left
	++	Increment	right to left
	--	Decrement	right to left
	sizeof	Storage requirement	right to left
3	*	Multiplication	left to right
	/	Division	left to right
	%	Remainder (modulo)	left to right
4	+	Addition	left to right
	-	Subtraction	left to right
5	>	Shift to right	left to right
	<	Shift to left	left to right
6	<	Less than	left to right
	>	Greater than	left to right
	<=	Less than or equal to	left to right
	>=	Greater than or equal to	left to right
7	==	Equal	left to right
	!=	Unequal	left to right
8		Bitwise AND	left to right
9	^	Bitwise EXOR	left to right
10	|	Bitwise OR	left to right
11	&&	Logical AND	left to right
12	||	Logical OR	left to right
13	?:	Conditional evaluation	right to left
14	=	Assignment	right to left
	#=	Abbreviated assignment # from (+, -, *, /, %, >>, <<, &, |, ~)	right to left
15	,	Separation of expressions	left to right

G. Storage Classes

Storage class	Validity	Duration
auto	Block	Block
extern	Program	Program
register	Block	Block
static (intern)	Block	Program
static (extern)	File	Program

H. Type Conversions

Rules:

1. `char` and `short` are always converted to `int` and `float` into `double`.

2. If after this conversion one of the operators should have the type `double`, the second operand and the result are also converted to `double`.

3. If a data type is `long`, all participating values are also transformed to `long`.

4. If an `unsigned` value is found among the operands, all values are converted to `unsigned`.

I. Modes for `fopen`

Lattice C

String	Create	Cut file	Read	Write	Append	Binary
"r"	no	no	yes	no	no	yes
"w"	yes	yes	no	yes	no	yes
"a"	yes	no	no	no	yes	yes
"r+"	no	no	yes	yes	no	yes
"w+"	yes	no	yes	yes	no	yes
"a+"	yes	no	yes	no	yes	yes
"ra"	no	no	yes	no	no	no
"wa"	yes	yes	no	yes	no	no
"aa"	yes	no	no	no	yes	no
"ra+"	no	no	yes	yes	no	no
"wa+"	yes	no	yes	yes	no	no
"aa+"	yes	no	yes	no	yes	no
"rb"	no	no	yes	no	no	yes
"wb"	yes	yes	no	yes	no	yes
"ab"	yes	no	no	no	yes	yes
"rb+"	no	no	yes	yes	no	yes
"wb+"	yes	no	yes	yes	no	yes
"ab+"	yes	no	yes	no	yes	yes

No conversions are made for binary files. If the file was opened as an ASCII file, which can be recognized by the "a" at the second place, all carriage returns (code 13 = `\r`) are eliminated and the character with the ASCII code (26) is converted to `EOF` (`-1`) during reading. During writing, the single line feed (`\n`), is converted to the character combination `\r\n`.

To differentiate the two modes, Lattice C uses an external `int` variable named `_fmode`. If the highest value bit is set (`_fmode & 0x8000`), the binary mode is used, or else the conversions indicated are performed.

Changes for Aztec

Aztec opens all files in binary. Aztec also offers the "x" and "x+" modes which open a file for writing. If the file doesn't yet exist, it is created. With "x+" the file can be read and written after opening.

Index

? conditional operator 163

Abbreviations 34, 95
Activate 237
Addresses 131
AND 165
Arguments 28, 105, 203
Arrays 115, 189
ASCII code 10, 84
auto 143, 155, 181
Aztec C 19, 39, 41, 47, 65, 210

Backslash 29, 85
<Backspace> key 18
BASIC 4, 48
Bit fields 175
Bit manipulation 165
Bit shifting 167
Bit-planes 251
Bitwise shift opperators 167
Braces 25, 28
break 123
Buffer 206

C language 4
Calculation 35, 61
case statement 189
cast operator 71
char 39, 57, 68, 111, 115, 133,136, 181, 203
char pointer 191
Character strings 39
CLI 10, 13, 17, 25, 203, 228
Comments 30, 258
Comparison operators 45
Compiler 3-5, 9, 27, 153
Complex data types 173
CON 219
Conditional operator 163
Console 218
Constant 177
continue 123, 125
Conversion program 82

CPU 206
create 213
<Ctrl><X> 18

Data types 105, 182, 231
De-referencing 133
Decimal notation 85
Declarations 99, 181
Decrement operator 97
define 174, 206
Definition 99
Direct access 216
Directives 229
do while loop 49, 58
DOS 257
double 67, 106, 136, 178

ED 10, 91
Editor 3, 9
else 33
End of paragraph 27
enum 177
EOF 209
Error checking 49
Errors 189, 196, 231
<Esc> key 18
Escape sequences 28
EXCLUSIVE OR 168
extern 99, 100, 145, 181

fgetc 211
File extension 10
File mode 208
float 67, 115, 136, 146 173, 181, 195
Floating point numbers 37
Floating point variables 67
Flowchart 9
fopen 207
for 48, 57, 123
Format specification 32, 77
fprintf 211, 220
fputc 211

fread 211
fscanf 208, 211
fseek 216
ftell 217
Function 12, 25, 28, 82, 105, 143, 195
fwrite 211

Gadget 235
getc 206
getchar 148, 207, 218
Global variables 155, 156
goto 169
Guru Meditation 37

Header files 91, 206, 236
Hexadecimal system 80, 81
High level languages 4

Icons 17, 227, 235
if 32-33, 45, 119
Increment operator 97
Index 58
Initialization 99, 184
int 31, 67, 106, 115, 133, 181, 231
Integer division 61
Integer variables 31
Integers 31
Interlace 252
Interpreted languages 4, 5
Intuition 235, 238
itoa 157

Keywords 25

Lattice C 13, 38, 41, 46, 65, 81, 91, 212
Library macros 198
Library 25, 238
Linefeeds 27
Linker 3, 9, 12, 27
Logical AND 52
Logical errors 13
Logical OR 52
LOGO 4
long 68, 69, 82, 115, 133, 181, 182, 217
Loops 45
lseek 216

Machine language 4
Macros 3, 195, 227
main function 25, 99, 105, 203
MAKE file 13, 203
Menus 235
Modula2 4
Modules 12
modulo 61, 157
Mouse 17, 235
Multi-dimensional arrays 117
Multiple assignment 101
Multiple dimensions 156
Multitasking 235

Nocarerefresh 237
Number conversion 82, 86

Object code 11
Octal system 80
One's complement 168
One-dimensional arrays 117
open function 213, 236
Operating system 235
OR 166

Pascal 4, 29
Pointer arrays 189
Pointers 131, 133, 173, 189, 235
Precedence 61, 136
Preferences 237
Preprocessor 89, 227, 229
Preprocessor directives 229
printf 22, 25, 27, 45, 75, 98, 124, 157, 189
Program format 27
putc 206
putchar 207
puts 189

RAW 220
Redirection 222
Referencing 133
Register 143, 146, 181
return 106
<Return> key 18, 40
Reverse 159
scanf 31, 75, 131, 134, 208, 218
Scanner 262
Screens 243
Semicolon 106
short 68, 69, 181
sizeof 164, 191, 212

Smart_Refresh 237
Source code 28, 96, 189
sprintf 220
square 106
Standard input/output 218
Statement block 34
static variables 144, 156, 181
stderr 223
stdin 223
stdio.h 91, 148, 163, 223
stdout 223
strcmp 154
strcpy 109, 135, 147, 190
Strings 57, 178
strlen 111
struct 173, 236
Structures 183, 235
switch 123, 125
Syntax errors 13
System crash 37

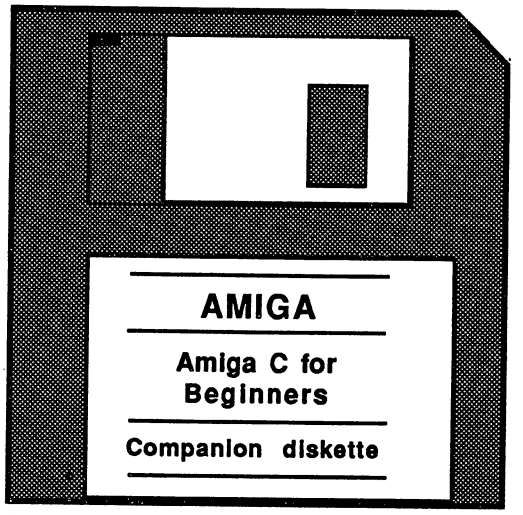
tell 217
text_arr 182
Tips and tricks 227
Transportability 6
Type conversion 70
typedef 178

Unbuffered input/output 213
unsigned 67, 175, 181
unsigned int 69
User-defined libraries 153

Variable declaration 105
Variables 31, 65, 143, 173
void 25, 105, 108

while 45, 83, 97, 98, 109, 123, 154
Window flags 237
windowdepth 237
windowdrag 237
windows 235
Windowsizing 237
Word processor 10
Workbench 227, 228

Companion Diskette



For your convenience, the program listings contained in this book are available on an Amiga formatted floppy diskette. You should order the diskette if you want to use the programs, but don't want to type them in from the listings in the book.

All programs on the diskette have been fully tested. You can change the programs for your particular needs. The diskette is available for $14.95 plus $2.00 ($5.00 foreign) for postage and handling.

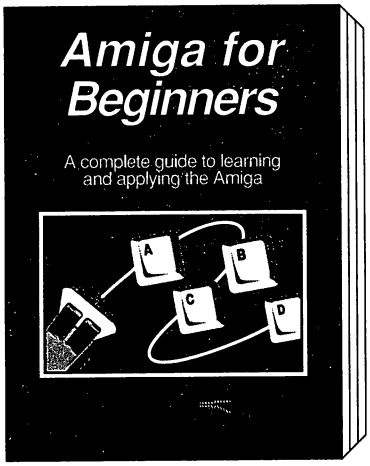
When ordering, please give your name and shipping address. Enclose a check, money order or credit card information. Mail your order to:

Abacus
5370 52nd Street SE
Grand Rapids, MI 49512

Or for fast service, call **616-698-0330.**
Credit Card orders only **1-800-451-4319.**

Amiga for Beginners — Vol.#1

A perfect introductory book if you're a new or prospective Amiga owner. **Amiga for Beginners** introduces you to Intuition (the Amiga's graphic interface), the mouse, windows, the versatile CLI. This first volume in our Amiga series explains every practical aspect of the Amiga in plain English. Clear, step-by-step instructions for common Amiga tasks. **Amiga for Beginners** is all the info you need to get up and running.



Topics include:

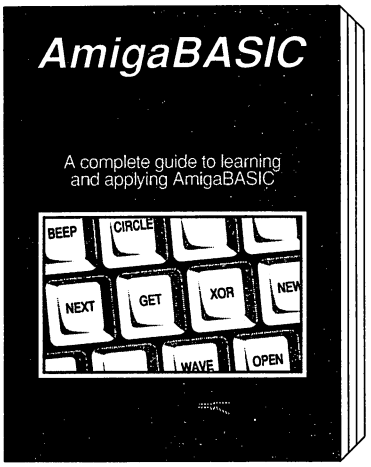
- Unpacking and connecting the Amiga components
- Starting up your Amiga
- Customizing the Workbench
- Exploring the Extras disk
- Taking your first step in AmigaBASIC programming language
- AmigaDOS functions
- Using the CLI to perform "housekeeping" chores
- First Aid, Keyword, Technical appendixes
- Glossary

ISBN 1-55755-021-2. Suggested retail price: $16.95

Companion Diskette not available for this book.

Amiga BASIC: Inside and Out — Vol.#2

Amiga BASIC: Inside and Out is the definitive step-by-step guide to programming the Amiga in BASIC. This huge volume should be within every Amiga user's reach. Every Amiga BASIC command is fully described and detailed. In addition, **Amiga BASIC: Inside and Out** is loaded with real working programs.



Topics include:

- Video titling for high quality object animation
- Bar and pie charts
- Windows
- Pull down menus
- Mouse commands
- Statistics
- Sequential and relative files
- Speech and sound synthesis

ISBN 0-916439-87-9. Suggested retail price: $24.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

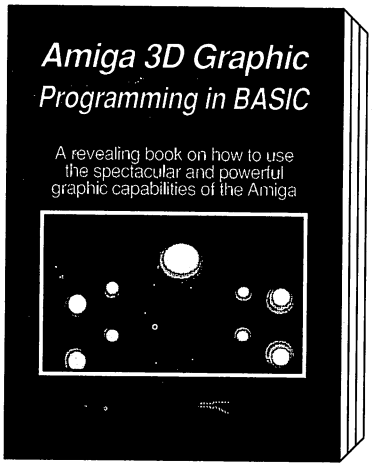
See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga 3D Graphic Programming in BASIC — Vol.#3

Amiga 3D Graphic Programming in BASIC- shows you how to use the powerful graphics capabilities of the Amiga. Details the techniques and algorithm for writing three dimensional graphics programs: ray tracing in all resolutions, light sources and shading, saving graphics in IFF format and more.

Topics include:

- Basics of ray tracing
- Using an object editor to enter three-dimensional objects
- Material editor for creating parameters of color, shading and mirroring of objects
- Automatic computation in different resolutions
- Using any Amiga resolution (low-res, high-res, interlace, HAM)
- Different light sources and any active pixel
- Save graphics in IFF format for later recall into any IFF compatible drawing program
- Mathematical basics for the non-mathematician



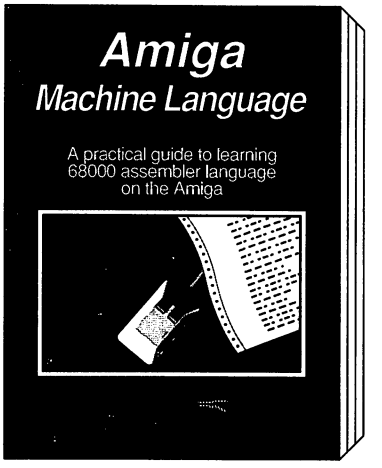
ISBN 1-55755-044-1. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

Amiga Machine Language — Vol.#4

Amiga Machine Language introduces you to 68000 machine language programming presented in clear, easy to understand terms. If you're a beginner, the introduction eases you into programming right away. If you're an advanced programmer, you'll discover the hidden powers of your Amiga. Learn how to access the hardware registers, use the Amiga libraries, create gadgets, work with Intuition and more.

- 68000 microprocessor architecture
- 68000 address modes and instruction set
- Accessing RAM, operating system and multitasking capabilities
- Details the powerful Amiga libraries for access to AmigaDOS
- Simple number base conversions
- Text input and output - Checking for special keys
- Opening CON: RAW: SER: and PRT: devices
- Menu programming explained
- Speech utility for remarkable human voice synthesis
- Complete Intuition demonstration program including Proportional, Boolean and String gadgets



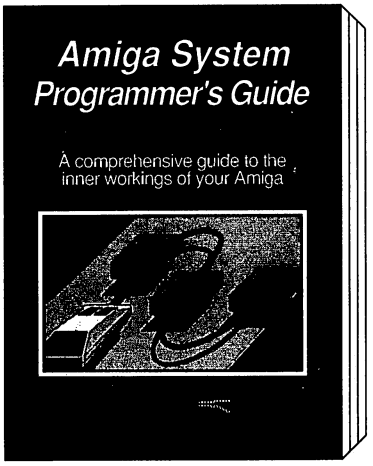
ISBN 1-55755-025-5. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga System Programmer's Guide — Vol.#6

Amiga System Programmer's Guide is a comprehensive guide to what goes on inside the Amiga in a single volume. Explains in detail the Amiga chips (68000, CIA, Agnus, Denise, Paula) and how to access them. All the Amiga's powerful interfaces and features are explained and documented in a clear precise manner.



Topics include:

- EXEC Structure
- Multitasking functions
- I/O management through devices and I/O request
- Interrupts and resource management
- RESET and its operation
- DOS libraries
- Disk management
- Detailed information about the CLI and its commands
- Much more—over 600 pages worth

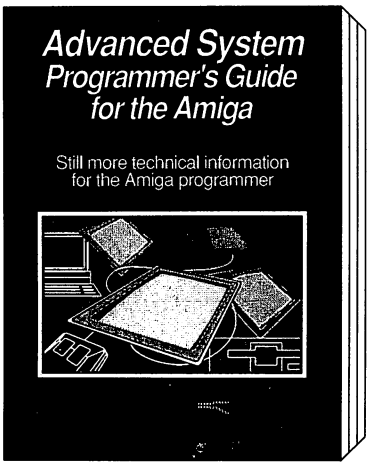
ISBN 1-55755-034-4. Suggested retail price: $34.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

Advanced System Programmer's Guide — Vol.#7

Advanced System Programmer's Guide for the Amiga - The second volume to our 'system programming' book. References all libraries, with basis and primitive sturctures. Devices: parallel, serial, printer, keyboard, gameport, input, console, clipboard, audio, translator, and timer trackdisk.

Some of the topics include:



- Interfaces- audio, video RGB, Centronics, serial, disk access, expansion port, and keyboard
- Programming hardware- memory organization, interrupts, the Copper, blitter and disk controller
- EXEC structures- Node, List, Libraries and Tasks
- Multitasking- Task switching, intertask communication, exceptions, traps and memory management
- I/O- device handling and requests
- DOS Libraries- functions, parameters and error messages
- CLI- detailed internal design descriptions

ISBN 1-55755-047-6. Suggested retail price: $34.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and readyto run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

AmigaDOS: Inside & Out

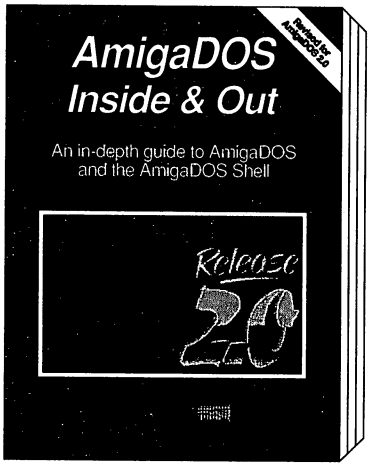
Revised for 2.0

Vol.#8

AmigaDOS: Inside & Out covers the insides of AmigaDOS from the internal design up to practical applications. **AmigaDOS Inside & Out** will show you how to manage Amiga's multitasking capabilities more effectively. There is also a detailed reference section which helps you find information in a flash, both alphabetically and in command groups. Topics include: Getting the most from the AmigaDOS Shell (wildcards and command abbreviations) • Script (batch) files - what they are and how to write them.

More topics include:

- AmigaDOS - Tasks and handling
- Detailed explanations of CLI commands and their functions
- In-depth guide to ED and EDIT
- Amiga devices and how the AmigaDOS Shell uses them
- Customizing your own startup-sequence
- AmigaDOS and multitasking
- Writing your own AmigaDOS Shell commands in C
- Reference for 1.2, 1.3 and 2.0 commands



ISBN 1-55755-041-7. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

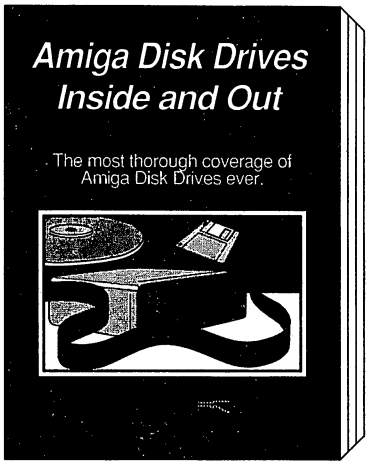
Amiga Disk Drives: Inside & Out

Vol.#9

Amiga Disk Drives: Inside & Out shows everything you need to know about Amiga disk drives. You'll find information about data security, disk drive speedup routines, disk copy protection, boot blocks, loading and saving programs, sequential and relative file organization and much more.

Topics include:

- Floppy disk operations from the Workbench and CLI
- DOS functions and operations
- Disk block types, boot blocks, checksums, file headers, hashmarks and protection methods
- Viruses and how to protect your boot block
- Trackdisk device: Commands and structures
- Trackdisk-task: Function and design
- MFM, GCR, track design, blockheader, datablocks, coding and decoding data, hardware registers, SYNC and interrupts



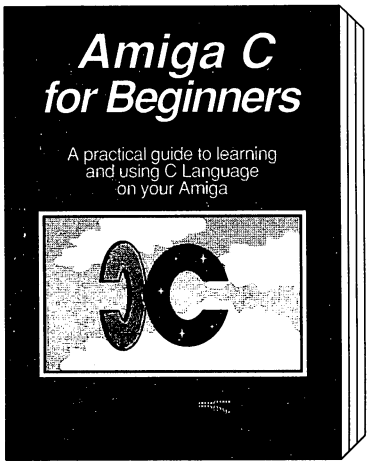
ISBN 1-55755-042-5. Suggested retail price: $29.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga C for Beginners — Vol.#10

Amiga C for Beginners is an introduction to learning the popular C language. Explains the language elements using examples specifically geared to the Amiga. Describes C library routines, how the compiler works and more.



Topics include:

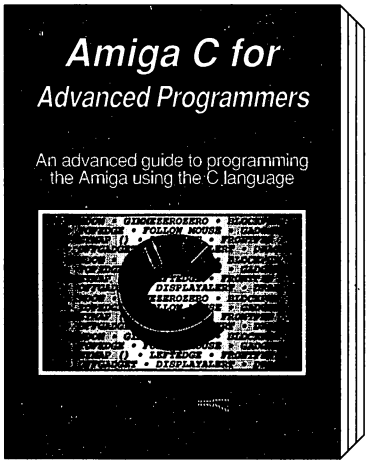
- Beginner's overview of C
- Particulars of C
- Writing your first program
- The scope of the language (loops, conditions, functions, structures)
- Special features of the C language
- Input/Output using C
- Tricks and Tips for finding errors
- Introduction to direct programming of the operating system (windows, screens, direct text output, DOS functions) Using the LATTICE and AZTEC C compilers

ISBN 1-55755-045-X. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

Amiga C for Advanced Programmers — Vol.#11

Amiga C for Advanced Programmers contains a wealth of information from the C programming pros: how compilers, assemblers and linkers work, designing and programming user friendly interfaces utilizing the Amiga's built-in user interface Intuition, managing large C programming projects, using jump tables and dynamic arrays, combining assembly language and C codes, using MAKE correctly. Includes the complete source code for a text editor.



Topics include:

- Using INCLUDE, DEFINE and CAST
- Debugging and optimizing assembler sources
- All about programming Intuition including windows, screens, pulldown menus, requesters, gadgets and more
- Programming the console device
- A professional editor's view of problems with developing larger programs
- Debugging C programs with different utilities

ISBN 1-55755-046-8. Suggested retail price: $34.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

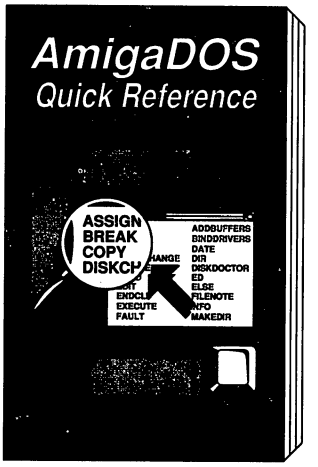
See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

AmigaDOS Quick Reference

AmigaDOS Quick Reference is an easy-to-use reference tool for beginners and advanced programmers alike. You can quickly find commands for your Amiga by using the three handy indexes designed with the user in mind. All commands are in alphabetical order for easy reference. The most useful information you need fast can be found including:

- All AmigaDOS commands described with examples including Workbench 1.3
- Command syntax and arguments described with examples
- CLI shortcuts
- CTRL sequences
- ESCape sequences
- Amiga ASCII table
- Guru Meditation Codes
- Error messages with their corresponding numbers

Three indexes for instant information at your fingertips! The **AmigaDOS Quick Reference** is an indispensable tool you'll want to keep close to your Amiga.



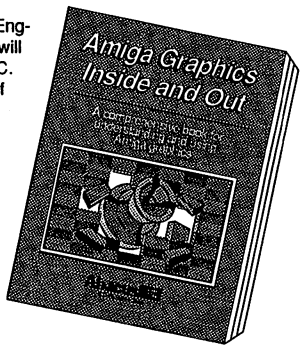
ISBN 1-55755-049-2. Suggested retail price: $9.95
Companion Diskette not available for this book.

Abacus Amiga Book Summary			
Vol.1	Amiga for Beginners	1-55755-021-2	$16.95
Vol.2	AmigaBASIC: Inside and Out	0-916439-87-9	$24.95
Vol.3	Amiga 3D Graphic Programming in BASIC	1-55755-044-1	$19.95
Vol.4	Amiga Machine Language	1-55755-025-5	$19.95
Vol.6	Amiga System Programmers Guide	1-55755-034-4	$34.95
Vol.7	Advanced System Programmers Guide	1-55755-047-6	$34.95
Vol.8	AmigaDOS: Inside and Out	1-55755-041-7	$19.95
Vol.9	Amiga Disk Drives: Inside and Out	1-55755-042-5	$29.95
Vol.10	'C' for Beginners	1-55755-045-X	$19.95
Vol.11	'C' for Advanced Programmers	1-55755-046-8	$24.95
Vol.13	Amiga Graphics: Inside & Out	1-55755-052-2	$34.95
Vol.14	Amiga Desktop Video Guide	1-55755-057-3	$19.95
Vol.15	Amiga Printers: Inside & Out w/ disk	1-55755-087-5	$34.95
Vol.16	Making Music on the Amiga w/disk	1-55755-094-8	$34.95
Vol.17	Best Amiga Tricks & Tips w/ disk	1-55755-107-3	$29.95
	AmigaDOS Quick Reference	1-55755-049-2	$9.95

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Books for the AMIGA

Amiga Graphics Inside & Out



The Amiga Graphics Inside & Out book will show you simply and in plain English the super graphic features and functions of the Amiga in detail. You will learn the graphic features that can be accessed from AmigaBASIC or C. The advanced user will learn graphic programming in C with examples of points, lines, rectangles, polygons, colors and more. Amiga Graphics Inside & Out contains a complete description of the Amiga graphic system - View, ViewPort, RastPort, bitmap mapping, screens, and windows.

Topics include:

- Accessing fonts and type styles in AmigaBASIC
- CAD on a 1024 x 1024 super bitmap, Using graphic library routines
- New ways to access libraries and chips from BASIC - 4096 colors at once, color patterns, screen and window dumps to printer
- Graphic programming in C - points, lines, rectangles, polygons,colors
- Amiga animation explained including sprites, bobs and AnimObs, Copper and blitter programming

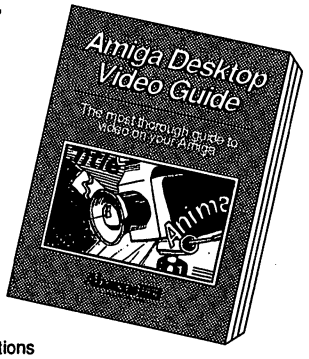
Volume 13 Suggested retail price $34.95 ISBN 1-55755-052-2

			GLN
Optional Diskette	**$14.95**	**#727**	**GLN**

Amiga Desktop Video Guide

The **Amiga Desktop Video Guide** is the most complete and useful guide to desktop video on the Amiga.

Amiga Desktop Video Guide covers all the basics – defining video terms, selecting genlocks, digitizers, scanners, VCRs, camera and connecting them to the Amiga.



Just a few of the topics you'll find described in this excellent book:

- The Basics of Video
- Genlocks
- Digitizers and Scanners
- Frame Grabbers/Frame Buffers
- How to connect VCRs, VTRs, and Cameras to the Amiga
- Animation
- Video Titling
- Music and Videos
- Home Video
- Advanced Techniques
- Using the Amiga to add or incorporate Special Effects to a video
- Tips on Paint, Ray Tracing, and 3-D Rendering in Commercial Applications

Volume 14 • Suggested Retail Price $19.95 • ISBN 1-55755-057-3

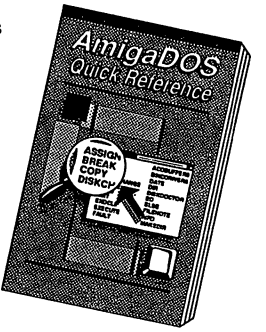
Save Time and Money!-Optional program disks are available for all our Amiga reference books (except Amiga for Beginners and AmigasDOS Quick Reference). Programs listed in the book are on each respective disk and saves countless hours of typing! $14.95

Books for the AMIGA

AmigaDOS Quick Reference Guide

AmigaDos Quick Reference Guide is an easy-to-use reference tool for beginners and advanced programmers alike. You can quickly find commands for your Amiga by using the three handy indexes designed with the user in mind. All commands are in alphabetical order for easy reference. The most useful information you need fast can be found- including:

- All AmigaDOS commands described, including Workbench 1.3
- Command syntax and arguments described with examples
- CLI shortcuts
- CTRL sequences
- ESCape sequences
- Amiga ASCII table
- Guru Meditation Codes
- Error messages with their corresponding numbers



Three indexes for quick information at your fingertips! The AmigaDOS Quick Reference Guide is an indispensable tool you'll want to keep close to your Amiga.

Suggested retail price US $9.95 ISBN 155755-049-2

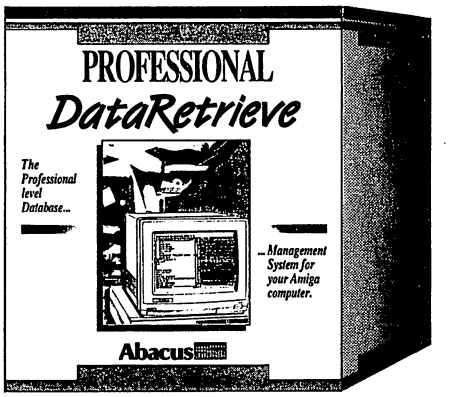
Abacus Amiga Books

Vol. 1 Amiga for Beginners	1-55755-021-2	$16.95
Vol. 2 AmigaBASIC Inside & Out	0-916439-87-9	$24.95
Vol. 3 Amiga 3D Graphic Programming in BASIC	1-55755-044-1	$19.95
Vol. 4 Amiga Machine Language	1-55755-025-5	$19.95
Vol. 5 Amiga Tricks & Tips	0-916439-88-7	$19.95
Vol. 6 Amiga System Programmers Guide	1-55755-034-4	$34.95
Vol. 7 Advanced System Programmers Guide	1-55755-047-6	$34.95
Vol. 8 AmigaDOS Inside & Out	1-55755-041-7	$19.95
Vol. 9 Amiga Disk Drives Inside & Out	1-55755-042-5	$29.95
Vol. 10 Amiga C for Beginners	1-55755-045-X	$19.95
Vol. 11 Amiga C for Advanced Programmers	1-55755-046-8	$34.95
Vol. 12 More Tricks & Tips for the Amiga	1-55755-051-4	$19.95
Vol. 13 Amiga Graphics Inside & Out	1-55755-052-2	$34.95
Vol. 14 Amiga Desktop Video Guide	1-55755-057-3	$19.95
AmigaDOS Quick Reference Guide	1-55755-049-2	$ 9.95

Abacus Products for *Amiga* computers

Professional DataRetrieve

The Professional Level Database Management System



Professional DataRetrieve, for the Amiga 500/1000/2000, is a friendly easy-to-operate professional level data management package with the features most wanted in a relational data base system.
Professional DataRetrieve has complete relational data mangagement capabilities. Define relationships between different files (one to one, one to many, many to many). Change relations without file reorganization.

Professional DataRetrieve includes an extensive programming laguage which includes more than 200 BASIC-like commands and functions and integrated program editor. Design custom user interfaces with pulldown menus, icon selection, window activation and more.
Professional DataRetrieve can perform calculations and searches using complex mathematical comparisons using over 80 functions and constants.

Professional DataRetrieve is a friendly, easy to operate programmable RELATIONAL data base system. **PDR** includes PROFIL, a programming language similar to BASIC. You can open and edit up to 8 files simultaneously and the size of your data fields, records and files are limited only by your memory and disk storage. You have complete interrelation between files which can include IFF graphics. NOT COPY PROTECTED. ISBN 1-55755-048-4

MORE features of **Professional DataRetrieve**

Easily import data from other databases....file compatible with standard **DataRetrieve**....supports multitasking...design your own custom forms with the completely integrated printer mask editor....includes PROFIL programming language that allows the programmer to custom tailor his database requirements...

MORE features of PROFIL include:

Open Amiga devices including the console, printer, serial and the CLI.
Create your own programmable requestors
Complete error trapping.
Built-in compiler and much, much more.

Suggested retail price: **$295.00**

Features

- Up to 8 files can be edited simultaneously
- Maximum size of a data field 32,000 characters (text fields only)
- Maximum number of data fields limited by RAM
- Maximum record size of 64,000 characters
- Maximum number of records disk dependent (2,000,000,000 maximum)
- Up to 80 index fields per file
- Up to 6 field types - Text, Date, Time, Numeric, IFF, Choice
- Unlimited number of searches and subrange criteria
- Integrated list editor and full-page printer mask editor
- Index accuracy selectable from 1-999 characters
- Multiple file masks on-screen
- Easily create/edit on-screen masks for one or many files
- User-programmable pulldown menus
- Operate the program from the mouse or the key board
- Calculation fields, Data Fields
  IFF Graphics supported
- Mass-storage-oriented file organization
- Not Copy Protected, NO DONGLE; can be in stalled on your hard drive

Selected Abacus Products for the *Amiga* computers

BeckerText

Powerful Word Processing Package for the Amiga

BeckerText ***Amiga*** is more than just a word processor.

BeckerText ***Amiga*** gives you all of the easy-to-use features found in our **TextPro** ***Amiga***, plus it lets you do a whole lot more. You can merge sophisticated IFF-graphics anywhere in your document. You can hyphenate, create indexes and generate a table of contents for your documents, automatically. And what you see on the **BeckerText** screen is what you get when you print the document—real WYSIWYG formatting on your Amiga.

But **BeckerText** gives you still more: it lets you perform calculations of numerical data within your documents, using flexible templates to add, subtract, multiply and divide up to five columns of numbers on a page. **BeckerText** can also display and print multiple columns of text, up to five columns per page, for professional-looking newsletters, presentations, reports, etc. Its expandable built-in spell checker eliminates those distracting typographical errors.

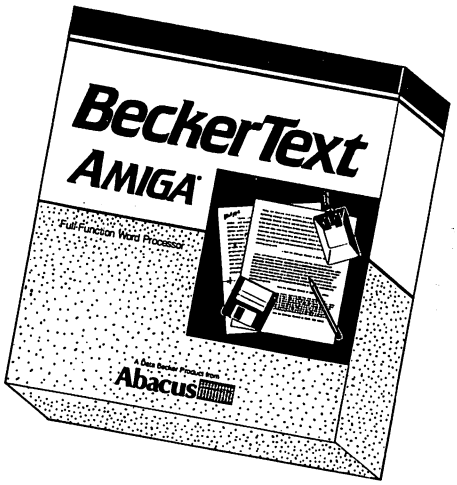
BeckerText works with most popular dot-matrix and letter-quality printers, and even the latest laser printers for typeset-quality output. Includes comprehensive tutorial and manual.

BeckerText gives you the power and flexibility that you need to produce the professional-quality documents that you demand.

When you need more from your word processor than just word processing, you need **BeckerText** ***Amiga***.

Discover the power of **BeckerText**.

Suggested retail price: $150.00



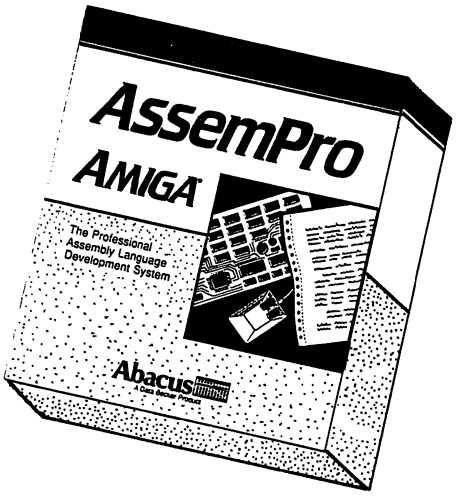
Features

- Select options from pulldown menus or handy shortcut keys
- Fast, true WYSIWYG formatting
- Bold, italic, underline, superscript and subscript characters
- Automatic wordwrap and page numbering
- Sophisticated tab and indent options, with centering and margin justification
- Move, Copy, Delete, Search and Replace
- Automatic hyphenation, with automatic table of contents and index generation
- Write up to 999 characters per line with horizontal scrolling feature
- Check spelling as you write or interactively proof document; add to dictionary
- Performs calculations within your documents—calculate in columns with flexible templates
- Customize 30 function keys to store often-used text and macro commands
- Merge IFF graphics into documents
- Includes ***BTSnap*** program for converting text blocks to IFF graphics
- C-source mode for quick and easy C language program editing
- Print up to 5 columns on a single page
- Adapts to virtually any dot-matrix, letter-quality or laser printer
- Comprehensive tutorial and manual
- Not copy protected

Selected Abacus Products for the *Amiga* computers

AssemPro

Machine Language Development System for the Amiga



Bridge the gap between slow higher-level languages and ultra-fast machine language programming: **AssemPro Amiga** unlocks the full power of the AMIGA's 68000 processor. It's a complete developer's kit for rapidly developing machine language/assembler programs on your Amiga. **AssemPro** has everything you need to write professional-quality programs "down to the metal": editor, debugger, disassembler & reassembler.

Yet **AssemPro** isn't just for the 68000 experts. **AssemPro** is easy to use. You select options from the dropdown menus or with shortcut keys, which makes your program development a much simpler process. With the optional Abacus book ***Amiga Machine Language*** (see page 3), **AssemPro** is the perfect introduction to Amiga machine language development and programming.

AssemPro also has the professional features that advanced programmers look for. Lots of "extras" eliminate the most tedious, repetitious and time-consuming m/l programming tasks. Like syntax error search/replace functions to speed program alterations and debugging. And you can compile to memory for lighting speed. The comprehensive tutorial and manual have the detailed information you need for fast, effective programming.

AssemPro Amiga offers more professional features, speed, sheer power, and ease of operation than any other assembler package we've seen for the money. Test drive your **AssemPro Amiga** with the security of the **Abacus 30-day guarantee.**

Suggested retail price: $99.95

Features

- Integrated Editor, Debugger, Disassembler and Reassembler
- Large operating system library
- Runs under CLI and Workbench
- Produces either PC-relocatable or absolute code
- Create custom macros for nearly any parameter (of different types)
- Error search and replace functions
- Cross-reference list
- Menu-controlled conditional and repeated assembly
- Full 32-bit arithmetic
- Advanced debugger with 68020 single-step emulation
- Written completely in machine language for ultra-fast operation
- Runs on any Amiga with 512K or more and Kickstart version 1.2
- Not copy protected

Machine language programming requires a solid understanding of the AMIGA's hardware and operating system. We do not recommend this package to beginning Amiga programmers

Abacus Products for *Amiga* computers

DataRetrieve

A Powerful Database Manager for the Amiga

Imagine a powerful database for your Amiga: one that's fast, has a huge data capacity, yet is easy to work with.

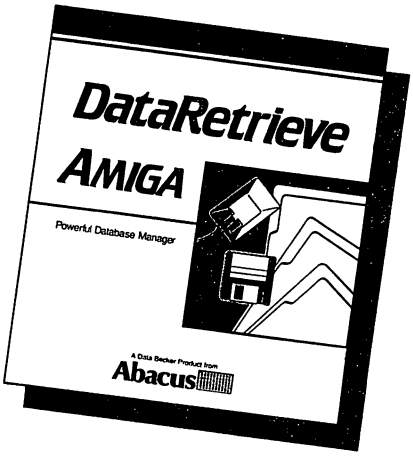
Now think **DataRetrieve** ***Amiga***. It works the same way as your Amiga—graphic and intuitive, with no obscure commands. You quickly set up your data files using convenient on-screen templates called masks. Select commands from the pulldown menus or time-saving shortcut keys. Customize the masks with different text fonts, styles, colors, sizes and graphics. If you have any questions, Help screens are available at the touch of a button. And **DataRetrieve's** 128-page manual is clear and comprehensive.

DataRetrieve is easy to use—but it also has professional features for your most demanding database applications. Password security for your data. Sophisticated indexing with variable precision. Full Search and Select functions. File sizes, data sets and data fields limited only by your memory and disk storage space. Customize up to 20 function keys to store macro commands and often-used text. For optimum access speed, **DataRetrieve** takes advantage of the Amiga's multi-tasking.

You can exchange data with **TextPro** ***Amiga***, **BeckerText** ***Amiga*** and other packages to easily produce form letters, mailing labels, index cards, bulletins, etc. **DataRetrieve** prints data reports to most dot-matrix & letter-quality printers.

DataRetrieve is the perfect database for your Amiga. Get this proven system today with the assurance of the Abacus **30-day MoneyBack Guarantee.**

Suggested retail price: **$79.95**



Features

- Select commands and options from the pulldown menus or shortcut keys
- Enter data into convenient screenmasks
- Enhance screen masks with different text styles, fonts, colors, graphics, etc.
- Work with 8 databases concurrently
- Define different field types: text, date, time, numeric & selection
- Customize 20 function keys to store macro commands and text
- Specify up to 80 index fields for *superfast* access to your data
- Perform simple or complex data searches
- Create subsets of a larger database for even faster operation
- Exchange data with other packages: form letters, mailing lists etc.
- Produce custom printer forms: index cards, labels, Rolodex®cards, etc. Adapts to most dot-matrix & letter-quality printers
- Protect your data with passwords
- Get Help from online screens
- Not copy protected

- Max. file size
- Max. data record size
- Max. data set
- Max. no. of data fields
- Max. field size

Limited only by your memory and disk space

Presenting...

Now Shipping

Abacus' AmigaDOS® Toolbox

A collection of essential, powerful, and easy-to-use tools for your Amiga.

Also included at no additional cost ($14.95 value)

Here's a new software package that every Amiga owner can use. Abacus' AmigaDOS Toolbox has the tools you need to make your Amiga computing easier and more productive. Whether you are a beginner or an advanced Amiga user you'll find the AmigaDOS Toolbox to be just what you've been looking for.

To order call Toll Free 1-800-451-4319

Abacus

Dept. L5, 5370 52nd Street S.E.
Grand Rapids, MI 49512
Phone: (616) 698-0330

Some of our best tools included are:

- DeepCopy- one of the fastest FULL disk copiers; copies many different formats.
- Speeder- a data speedup utility (more than 300%) -not a disk cache.
- BTSnap- a screen grabber deluxe.
- Diskmon- a full-featured disk editing tool.
- Fonts- eleven new originals you can use in your Amiga text.

...and many additional tools that every Amiga owner can use. Bought individually, equivalent software could cost up to $200.

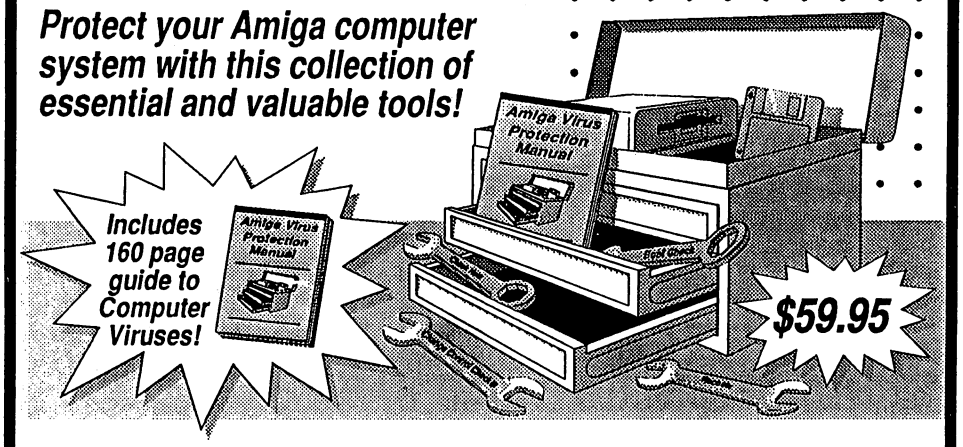
Amiga and AmigaDOS are registered trademarks of Commodore-Amiga Inc.

Presenting...

Abacus' Amiga® Virus Protection Toolbox

Now Shipping

Protect your Amiga computer system with this collection of essential and valuable tools!



The Virus Protection Toolbox describes how computer viruses work; what problems viruses cause; how viruses invade the Libraries, Handler and Devices of the operating system; preventive maintenance; how to cure infected programs and disks. Works with Workbench 1.2 and 1.3!

Some of our best tools included are:

- **Boot Check-** to prevent startup viruses.
- **Recover-** to restore the system information to disk.
- **Change Control Checker-** to record modifications to important files.
- **Check New-** to identify new program and data files.

Order now or call for your Free pamphlet "What you should know about Computer Viruses" (while supplies last)

Abacus

5370 52nd Street S.E.
Grand Rapids, MI 49512
Available at your local dealer or

Order Toll Free 1-800-451-4319

Amiga is a registered trademark of Commodore-Amiga Inc.

Amiga® C for Beginners

The C language is fast becoming the programming language of choice for Amiga users worldwide. **Amiga C for Beginners** is an introductory guide for the Amiga user who wants to learn and understand this popular language. He'll learn the basic and intermediate programming techniques and without a lot of technical jargon! **Amiga C for Beginners** shows you in plain English the language elements using examples specifically tailored to the Amiga.

Amiga C for Beginners explains the basic structure and peculiarities of each C statement and function. And it shows the user how to use many of the features of the Amiga's operating system - windows, screen, text output, DOS functions and more. **Amiga C for Beginners** gives you all you need to immediately start programming in C with your Amiga.

Amiga C for Beginners describes C library routines - C language structures - how a C compiler works - using C with your Amiga and much, much more.

US $19.95

ISBN 1-55755-045-X

9 781557 550453

A Practical guide to learning and using C on your Amiga

Topics include:

- Beginner's overview of C
- Writing your first C program
- Special features of the C language
- Important routines in the C libraries
- Input and Output using C
- The scope of the C language revealed (loops, conditions, functions, structures)
- Tricks & Tips for finding errors
- Using two of the most popular C compilers
- And much more

Optional Program Diskette available:
Contains every program listed in the book– complete, error-free and ready to run! Saves you hours of typing in the programs.

Abacus
5370 52nd Street SE, Grand Rapids, MI 49512

Amiga is a registered trademark of Commodore-Amiga Inc.